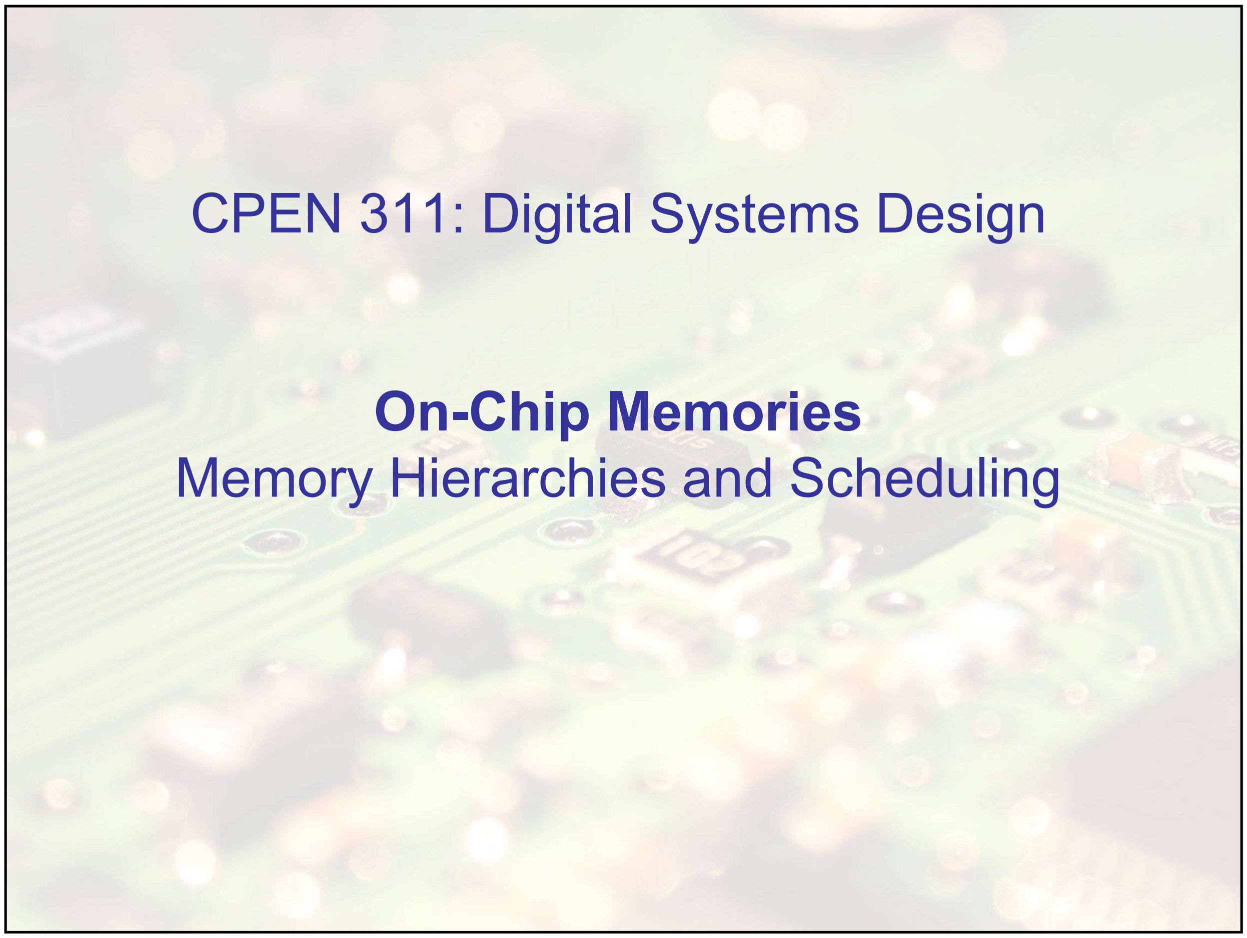


CPEN 311: Digital Systems Design

On-Chip Memories
Memory Hierarchies and Scheduling

1

Learning Objectives for Slide Set

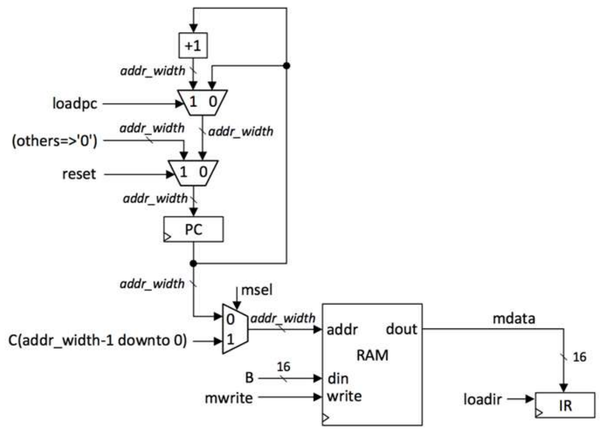
After this slide set, you will be able to:

1. Understand the differences, advantages and disadvantages of off-chip memory and on-chip memory
2. Understand three ways on-chip memory can be implemented in an FPGA
3. Understand the notion of memory ports, and be able to speak about single-port, dual-port and true-dual port memory configurations
4. Understand how the number and type of ports impacts overhead
5. Be able to schedule an algorithm subject to the availability of memory ports
6. Understand typical on-chip memory timing
7. Be able to write Verilog descriptions containing memory

Memories

In CPEN211, you learned about memories, including address ports, data ports, write enable.

Depending on when you took CPEN 211, you might have implemented a processor containing a memory, like this:



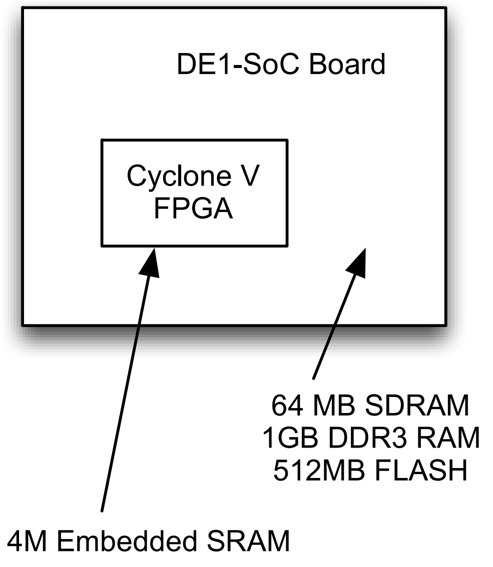
Our focus will be different: using memories to implement algorithms
- They dictate scheduling of operations (1 access per cycle per port)
- Often, memory are the limiting factor in performance

TL;DR

be very careful about modules that use memory

- can’t write to same port multiple times
    - if you do and use implicit memory instances, you will get “memories” from FFs
- can’t load (these) memories within one cycle
- need to design your memory client carefully

On-Chip vs. Off-Chip



Local storage on-chip

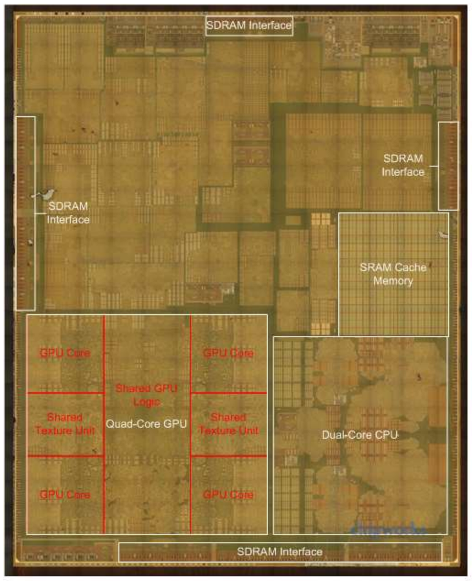
For larger datasets, use on-board SDRAM

For much larger storage, connect external memories, or interface to internet (and use cloud storage)

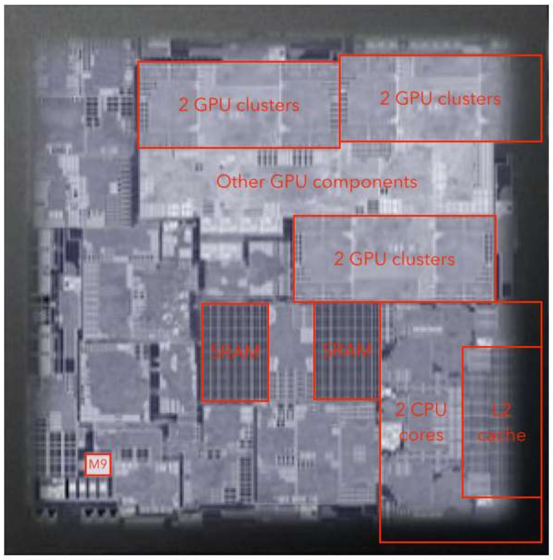
In this slide set, we will focus mostly **on on-chip** memory

On-Chip (Embedded) Memory

In a custom chip, you can instantiate a memory block that is exactly the size you need. Often done using a **memory generator**



A8 Processor

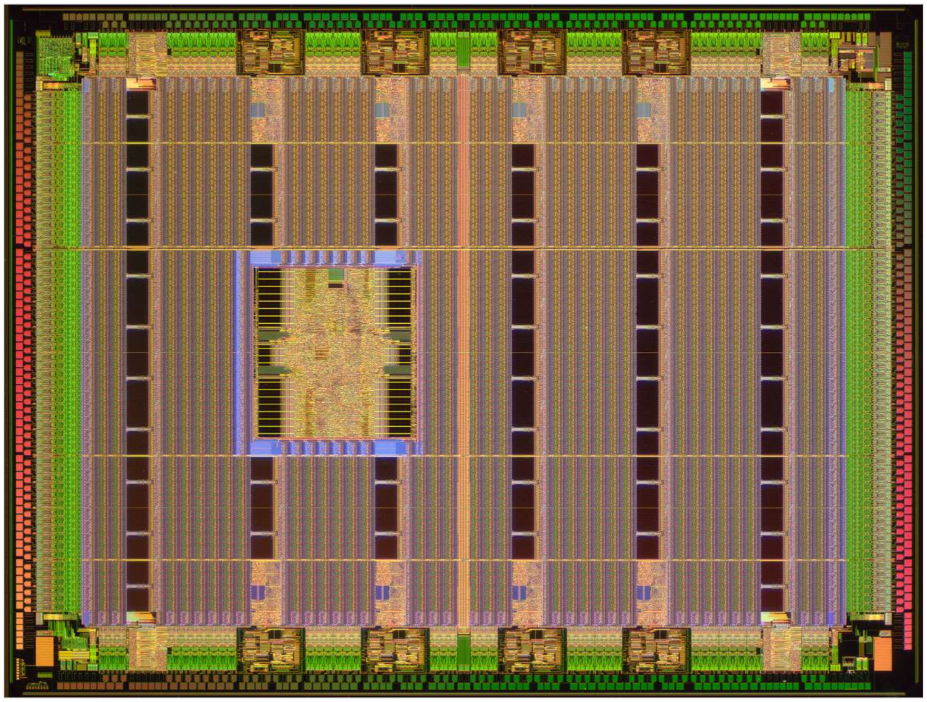


A9 Processor

Page 6

On-Chip (Embedded) Memory

In an FPGA, you need to use pre-fabricated memory resources



Page 7

7

Next topic: Ways to implement storage on an FPGA

A) Embedded Memory Blocks

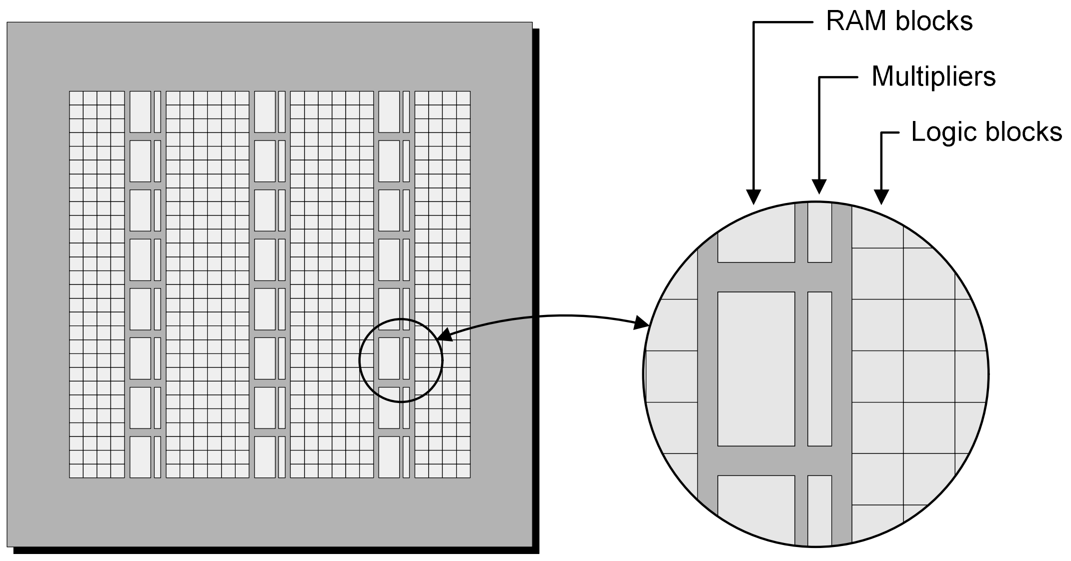
B) Flip-Flops in the logic fabric

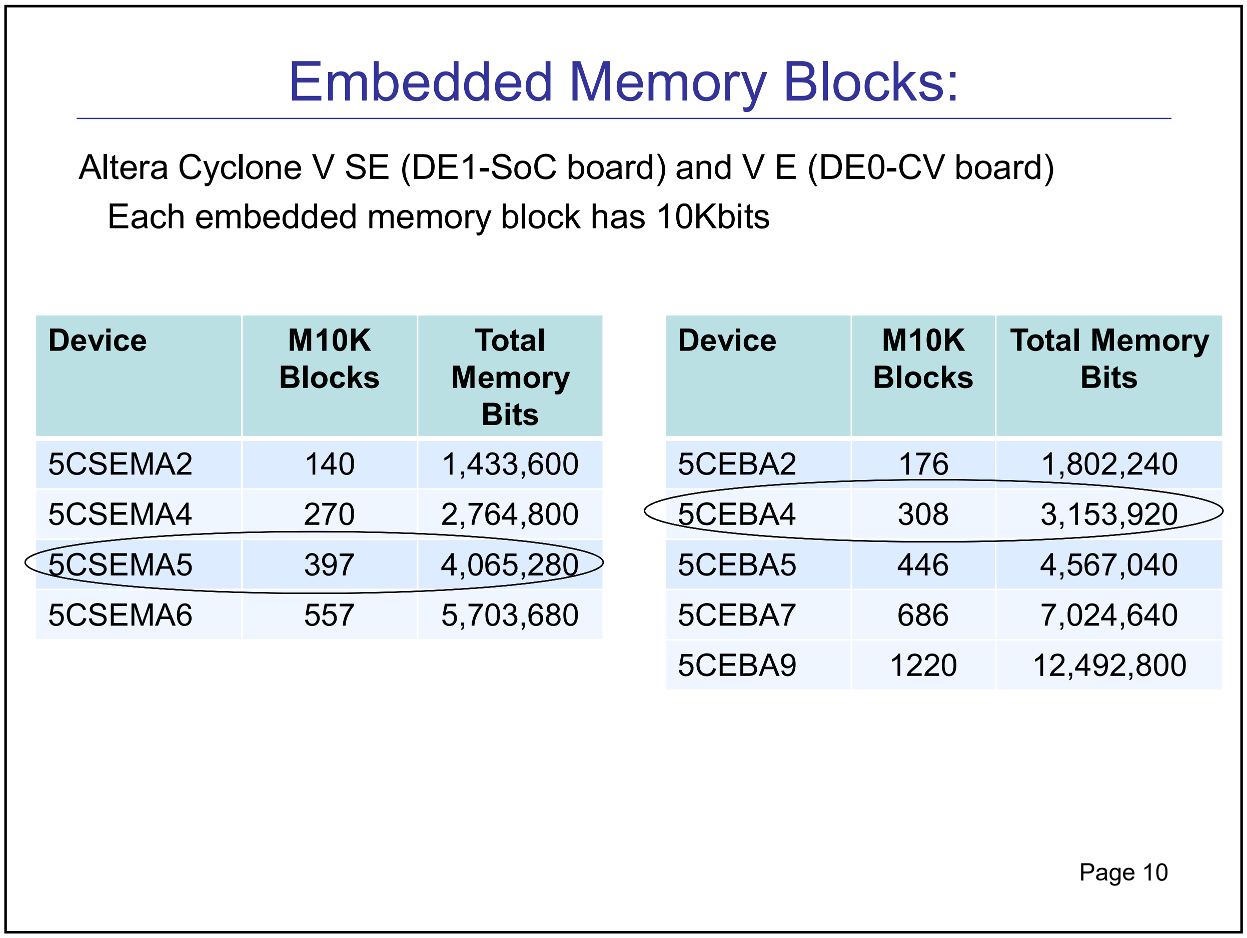
C) Configuration Storage (only on certain FPGA devices, not ours)

This is important because these methods have very different overheads and constraints.

A) First way is Embedded Memory Blocks:

- FPGA Manufacturer includes memory blocks (at the expense of logic blocks):





Embedded Memory Blocks:

Altera Cyclone V SE (DE1-SoC board) and V E (DE0-CV board)

Each embedded memory block has 10Kbits

Device	M10K Blocks	Total Memory Bits
5CSEMA2	140	1,433,600
5CSEMA4	270	2,764,800
5CSEMA5	397	4,065,280
5CSEMA6	557	5,703,680

Device	M10K Blocks	Total Memory Bits
5CEBA2	176	1,802,240
5CEBA4	308	3,153,920
5CEBA5	446	4,567,040
5CEBA7	686	7,024,640
5CEBA9	1220	12,492,800

Page 10

10

Advantages of using embedded memory blocks:

- More efficient for large memories
    - In our chip up to 484kB / 4Mb on-chip storage
- Address decoders, muxes, etc. already implemented as part of the memory blocks (don't need to implement these using logic blocks)

Disadvantages of using embedded memory blocks:

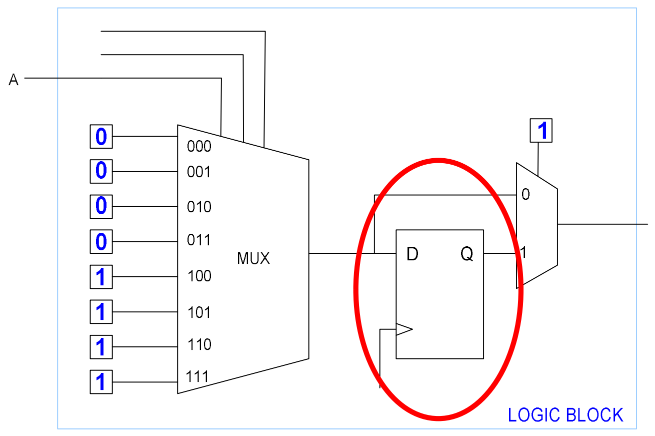
- Less flexible
- Registers are faster for small sizes

For > 1kb, embedded memories are the way to go

Flip-Flop Memory:

B) Second way to implement memory in an FPGA:

Each logic element has a flip-flop



DE1-SoC (5CSEMA5) has 128k flip-flops

- Not a very efficient way to implement large memories!

Page 12

Advantages of using flip-flops to implement memory:

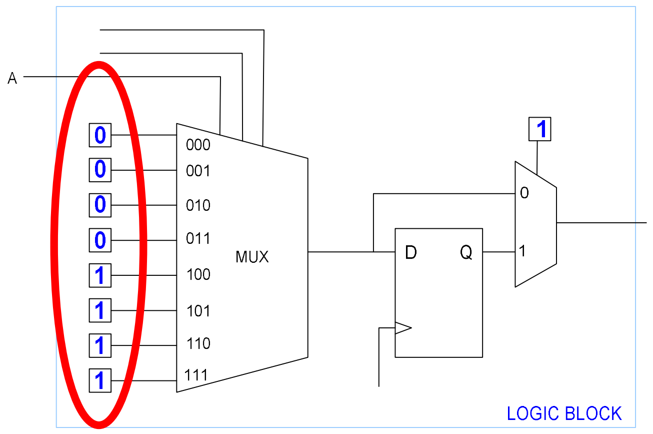
- Very flexible, you can combine registers however you like
- Fast connections between logic and memory, since registers are in the logic fabric

Disadvantages of using flip-flops to implement memory:

- Each bit of memory takes a logic block (quickly run out)
- You need to construct address decoders, output muxes, etc., out of general purpose logic. These also are large and will quickly fill up your chip (more on this coming…)

For <1kb, registers are the way to go.

C) Third way to implement memory in an FPGA is configuration storage:



Configuration bits are actually memory bits! Some FPGAs (not ours) lets you use these as user memory.

We won't talk about this option any more in this slide set.

Next topic: Read and Write Port Configurations

a) Fixed Read and Write ports in Embedded Memories
b) Creating Read and Ports using Flip-Flop Memories

This is important because the read and write port configuration impacts how you can schedule your algorithm in hardware.

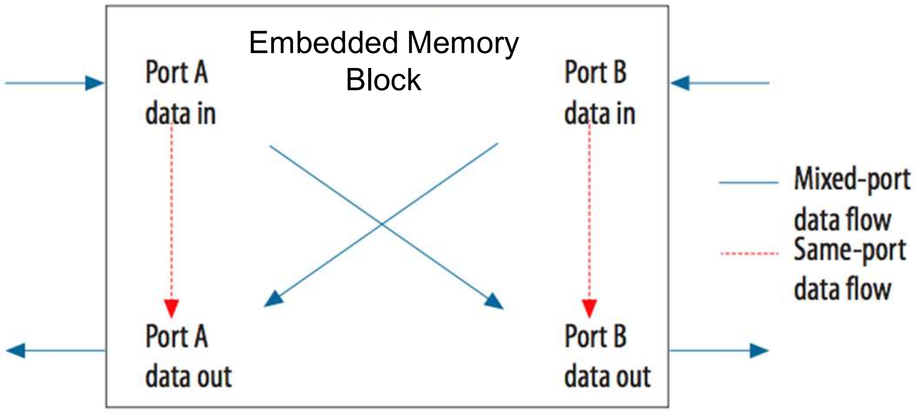
Embedded Memory Blocks

Read and Write Ports

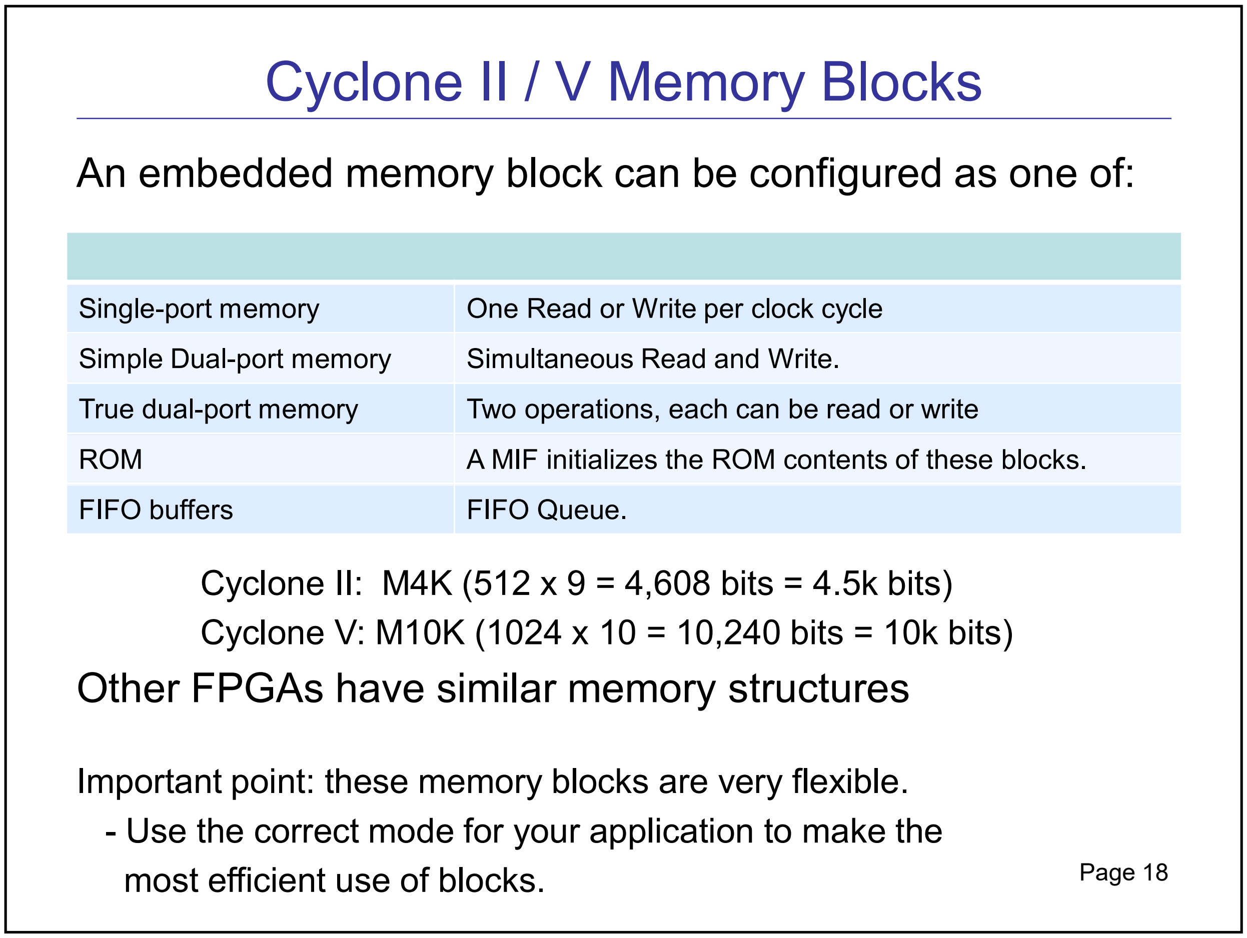
Cyclone II / V Memory Blocks

Memory blocks are highly configurable

Physically they are built like this:



Ports A & B share one physical memory, mild constraints on crossing
Logically, they can be configured into several modes (see next slide)



Cyclone II / V Memory Blocks

An embedded memory block can be configured as one of:

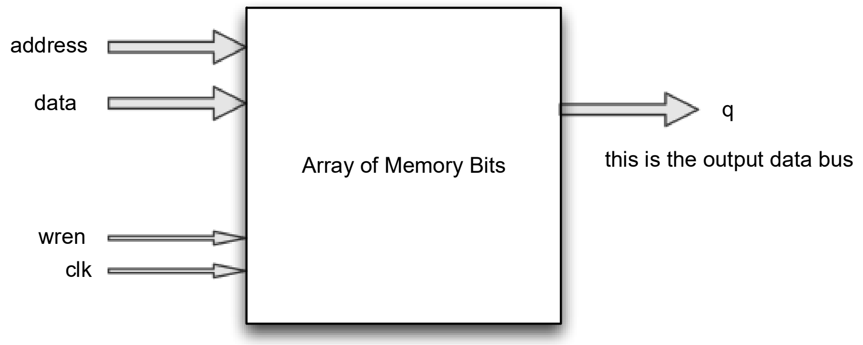
Single-port memory	One Read or Write per clock cycle
Simple Dual-port memory	Simultaneous Read and Write.
True dual-port memory	Two operations, each can be read or write
ROM	A MIF initializes the ROM contents of these blocks.
FIFO buffers	FIFO Queue.

Cyclone II: M4K (512 x 9 = 4,608 bits = 4.5k bits)
Cyclone V: M10K (1024 x 10 = 10,240 bits = 10k bits)

Other FPGAs have similar memory structures

Important point: these memory blocks are very flexible.
- Use the correct mode for your application to make the most efficient use of blocks.

Single-Port Memory: Generic View



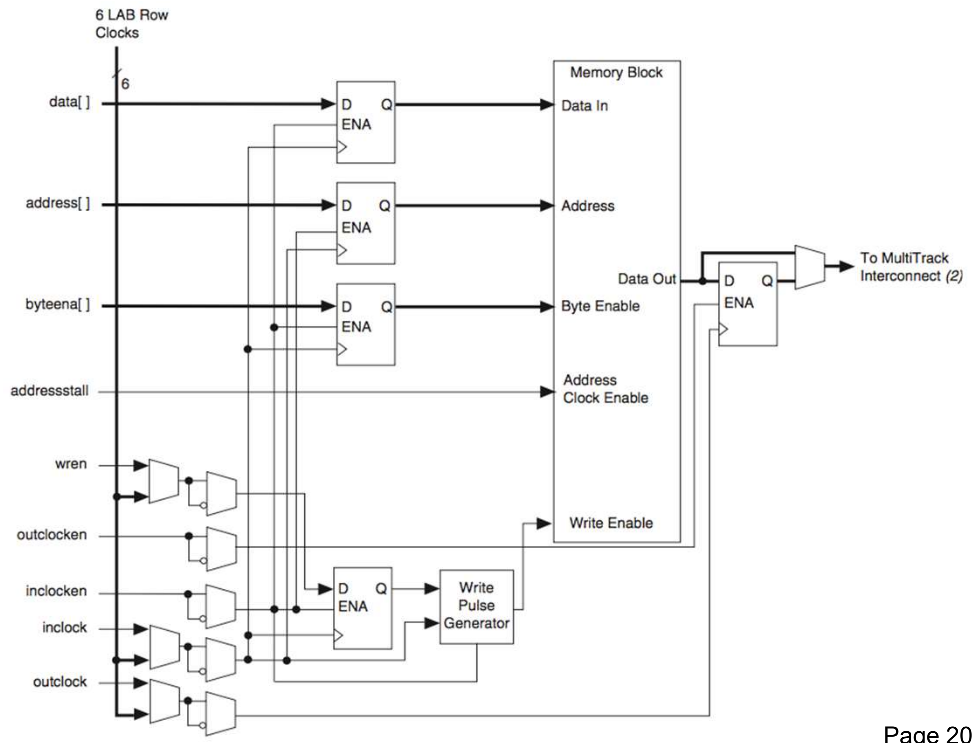
Each cycle, you can either write or read a single word from memory (wren determines whether you are writing or reading).

Write: put address on address bus, data on data bus and assert wren

Read: put address on address bus, de-assert wren, data appears on q

Page 19

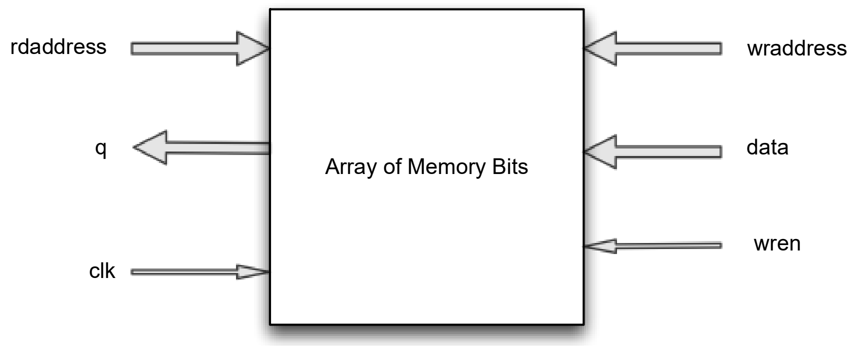
Single-Port Memory: details



From Altera, Cyclone II handbook

Page 20

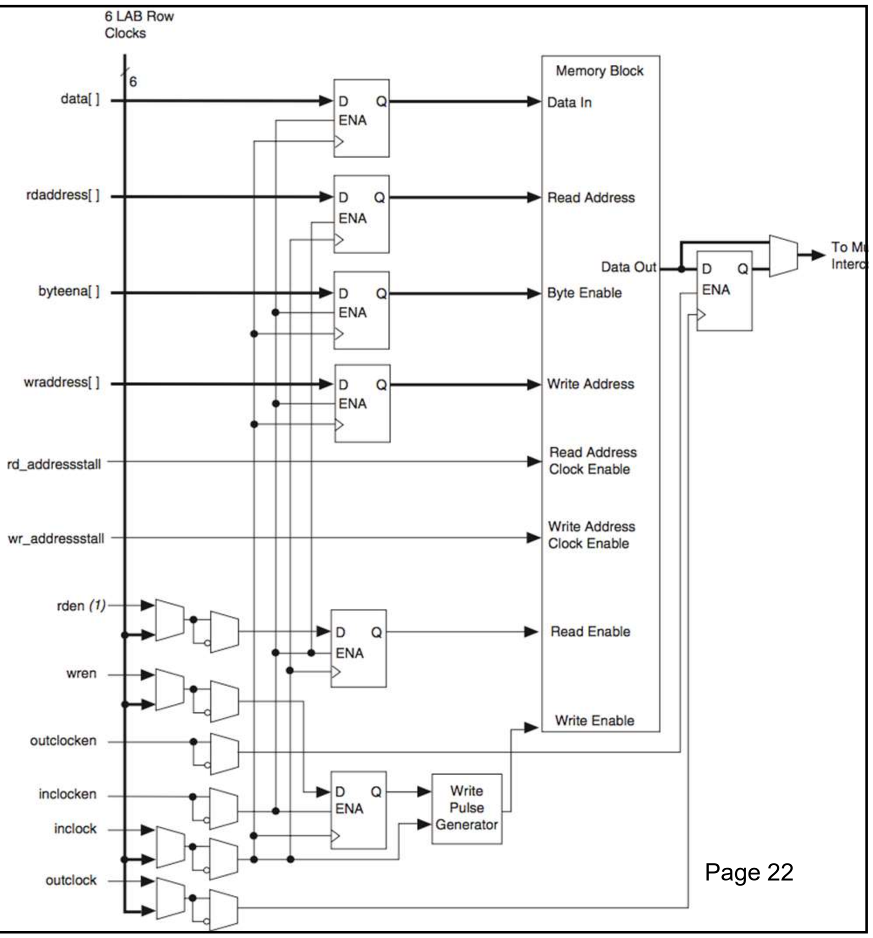
Simple Dual-Port Memory: Generic View



Each cycle, you can write a word *and* read a word
- Normally, addresses will be different
- If you write and read the same address at the same time, there are several possible policies (read old, read new)
- Some memories allow for separate clocks

This sort of memory is good for implementing FIFOs.

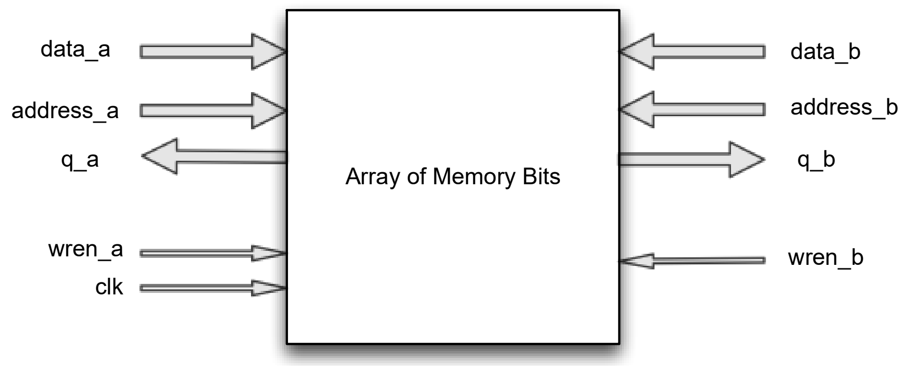
Simple Dual-Port Memory: details



From Altera, Cyclone II handbook

Page 22

True Dual-Port Memory: Generic View

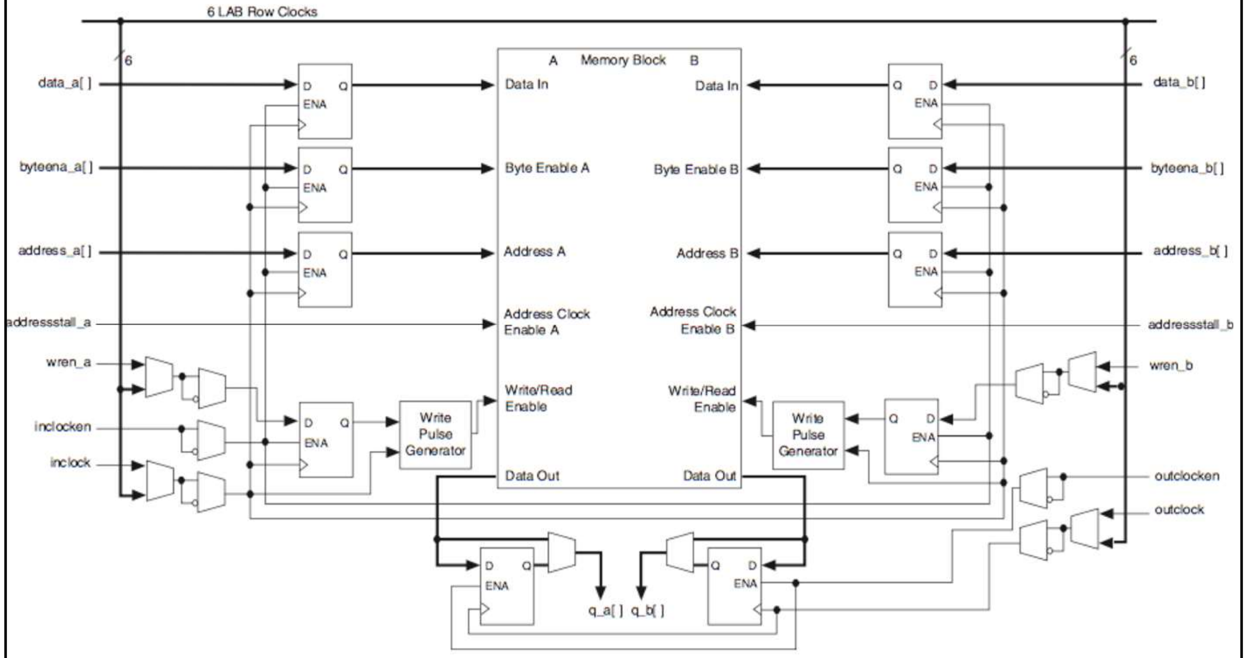


Each cycle, you can perform two access, either of which can be a write or read

- Some systems allow two different clocks

Page 23

True Dual-Port Memory: details



From Altera, Cyclone II handbook

Page 24

24

When you use embedded memory blocks, this port circuitry is included in each memory block already.

What if we want to use flip-flop memory rather than the embedded blocks?

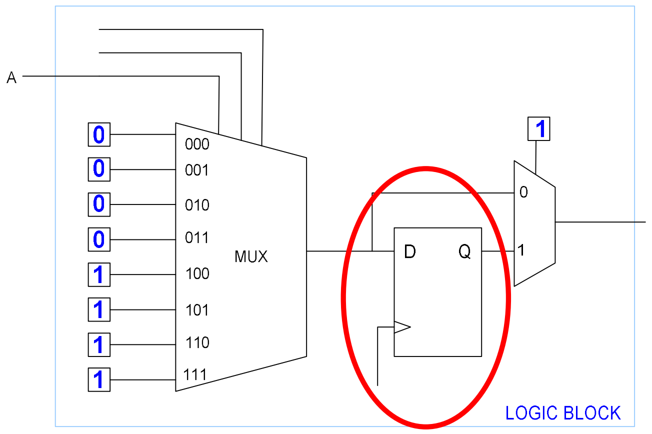
Flip-Flop Memory

Read and Write Ports

Page 26

Flip-Flop Memory: Ports

What if you want to use flip-flop memory?



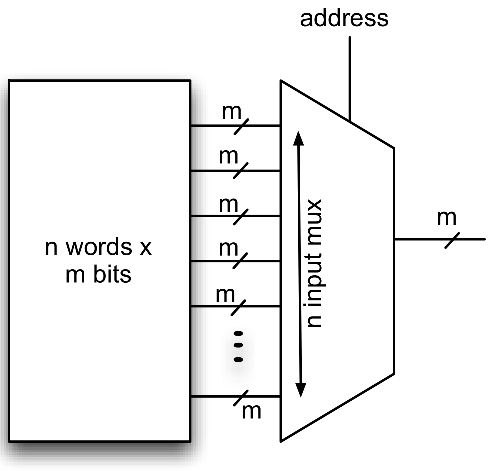
Read and write ports need to be implemented out of general purpose logic

GOOD News: you can implement exactly the port structure you want.
BAD News: ports use a lot of logic, very inefficient use of your chip

Page 27

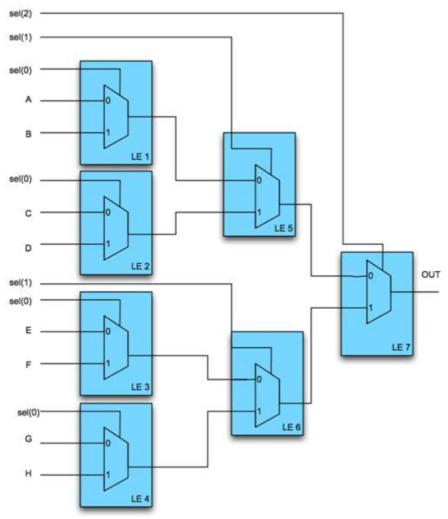
Flip-Flop Memory: Read Ports

Consider implementing a read-port out of general-purpose logic:



How big is this mux?

Consider building an 8-1 mux



In this case, you could do 8 input, 1 bit mux in 7 logic blocks

4+2+1 = 7

In general, you can do a n input, 1 bit mux in n-1 logic blocks

(n/2) + (n/4) + … 1 = n-1

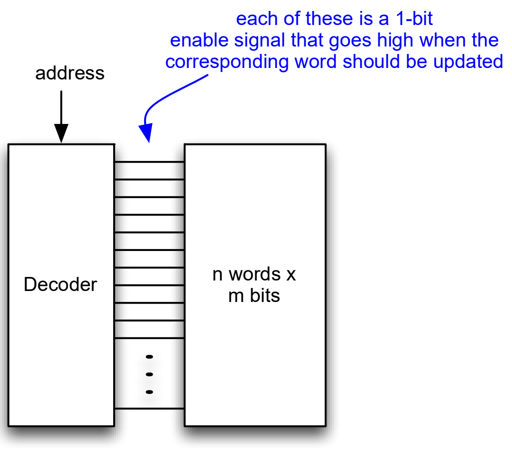
If the mux is m bits wide, you would need about m*(n-1) logic blocks

Example: a 256 bit x 8 bit memory would need 2040 logic blocks just to implement one read port!

Flip-Flop Memory: Write Ports

The size of each write port is a bit more tricky to work out, but we can approximate it:

- The heart of the write port is a decoder with n outputs. Each output goes high when the corresponding word is being written to


each of these is a 1-bit enable signal that goes high when the corresponding word should be updated
address
Decoder
n words x m bits

This decoder would need at least n logic blocks, since each logic block has only one output.

So a 256x8 bit memory would need at least 256 logic blocks to implement the write port (actually more than this)

Flip-Flop memories use a lot of logic blocks:

Consider a 256 x 8 bit memory with one read port and one write port:

Logic blocks required to implement storage: 256x8 = 2048

Logic blocks required to implement read port: 2040

Logic blocks required to implement write port: 256

Total is 4344 logic blocks. FPGA on DE2 has only has 33,216 logic blocks!

We can add more ports. Suppose we want 4 read ports (because we want to read 4 words each cycle):

2048 + 4*2040 + 256 = 10464 logic blocks

And this is a small memory!

Page 31

Moral: using flip-flop memories uses a lot of logic blocks.

Danger: Suppose you decide you want to use embedded memories. It turns out that if you don't write your Verilog "just right", the synthesis tool will give you flip-flop memories.

Why is this? That is the next topic.

Next topic: Describing Memories in Verilog

a) Explicit Instantiation

b) Inferred Memories

.

Including Memories in Verilog

1. **Explicit Instantiation:**
   - Explicitly indicate that you are including a memory using a pre-defined Altera macro. You can specify the number of ports, size, width, output registers, etc.
   - Can be done either directly with a port map or through a Wizard

2. **Inferred Memories:**
   - Write the behaviour of a memory using behavioural code and expect the tool to figure out that you want a memory (use the "array" construct in Verilog)
   - Inferring to embedded RAMs is tricky: you have to get it right
   - Inferring to flip-flop memories is relatively easy

Explicit Representation

Most systems contain macros that you can use to explicitly indicate a memory.

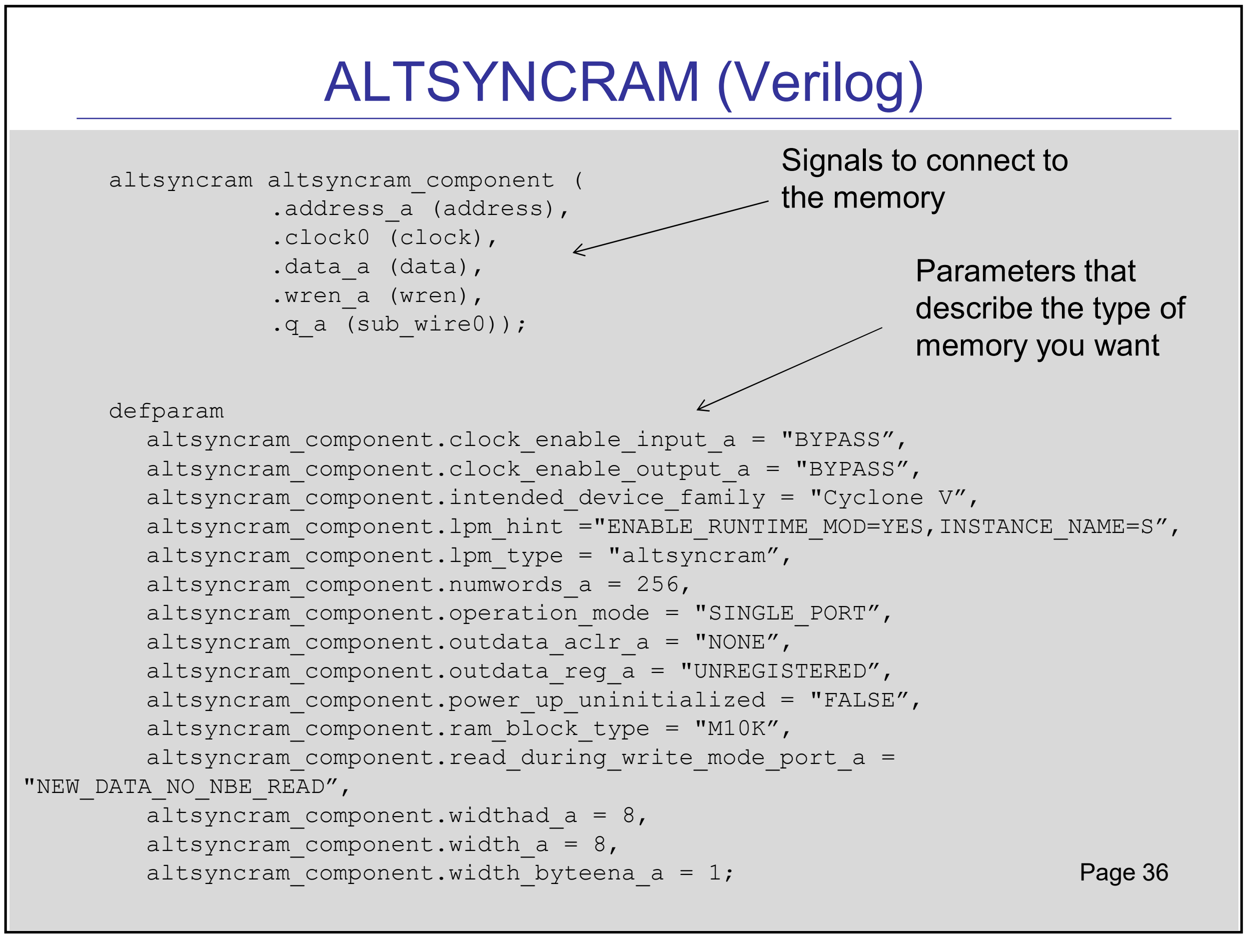
In Altera's case, you can use ALTSYNCRAM

Details on ALTSNYCRAM are at

http://quartushelp.altera.com/14.1/mergedProjects/hdl/mega/mega_file_altsynch_ram.htm

You can also use LPM_RAM_DQ, however, it has been depreciated (still available for backwards compatibility).

Page 35



ALTSYNCRAM (Verilog)

Signals to connect to the memory

Parameters that describe the type of memory you want

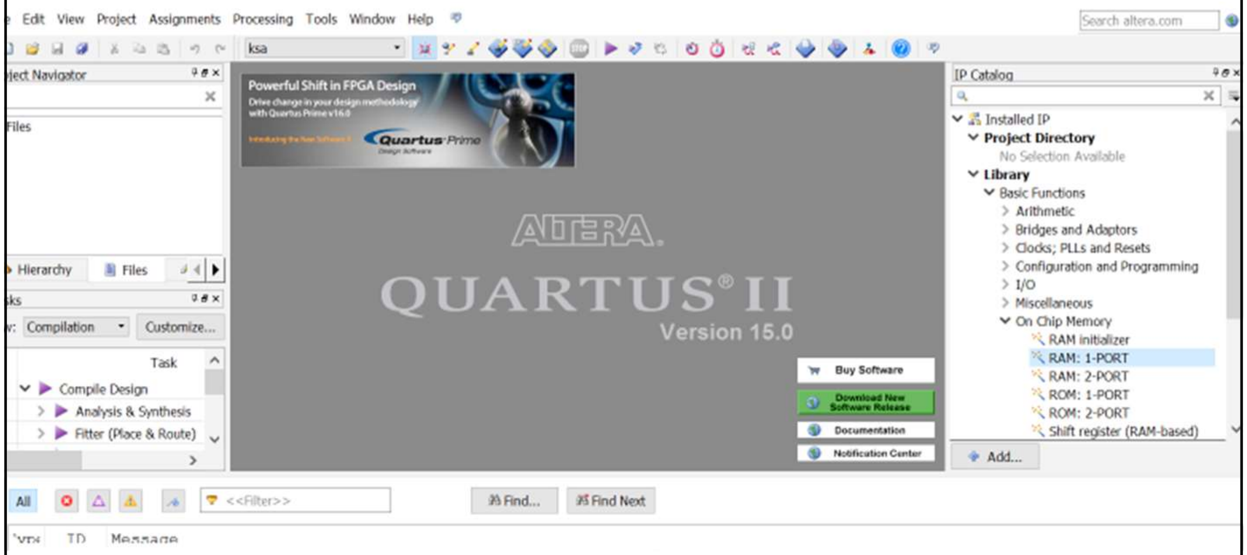
```
      altsyncram altsyncram_component (
                 .address_a (address),
                 .clock0 (clock),
                 .data_a (data),
                 .wren_a (wren),
                 .q_a (sub_wire0));


      defparam
         altsyncram_component.clock_enable_input_a = "BYPASS”,
         altsyncram_component.clock_enable_output_a = "BYPASS”,
         altsyncram_component.intended_device_family = "Cyclone V”,
         altsyncram_component.lpm_hint ="ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=S”,
         altsyncram_component.lpm_type = "altsyncram”,
         altsyncram_component.numwords_a = 256,
         altsyncram_component.operation_mode = "SINGLE_PORT”,
         altsyncram_component.outdata_aclr_a = "NONE”,
         altsyncram_component.outdata_reg_a = "UNREGISTERED”,
         altsyncram_component.power_up_uninitialized = "FALSE”,
         altsyncram_component.ram_block_type = "M10K”,
         altsyncram_component.read_during_write_mode_port_a =
"NEW_DATA_NO_NBE_READ”,
         altsyncram_component.widthad_a = 8,
         altsyncram_component.width_a = 8,
         altsyncram_component.width_byteena_a = 1;
```

Page 36

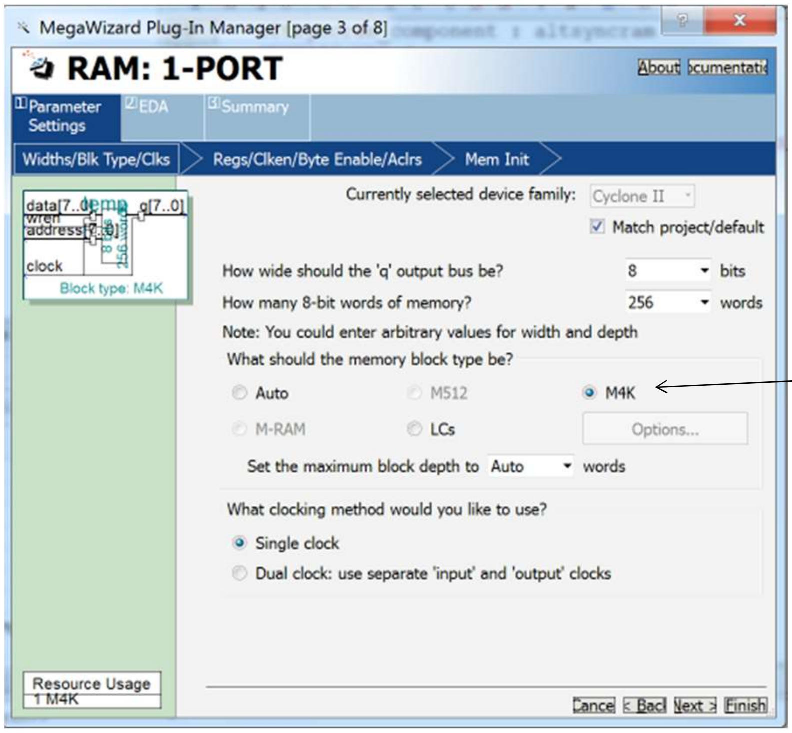
36

How to generate these?



Page 37

37



MegaWizard Plug-In Manager [page 3 of 8]

RAM: 1-PORT

Currently selected device family: Cyclone II

[✓] Match project/default

How wide should the 'q' output bus be? 8 bits

How many 8-bit words of memory? 256 words

Note: You could enter arbitrary values for width and depth

What should the memory block type be?

Auto / M512 / [✓ M4K] / M-RAM / LCs

Set the maximum block depth to Auto words

What clocking method would you like to use?

[✓ Single clock]

Dual clock: use separate 'input' and 'output' clocks

Resource Usage: 1 M4K

Choose whether you want an embedded memory or flip-flop memory

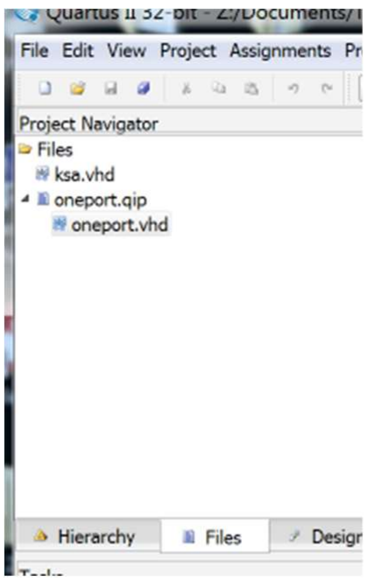
Goes through 8 pages…. at one point you can indicate VHDL or Verilog. Doesn't matter which

Page 38

Creates a lower level file with a module that you can include in your design:

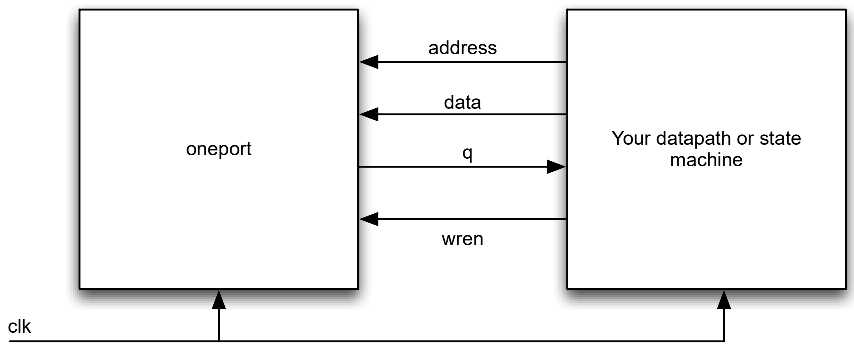
```
module oneport (
        address,
        clock,
        data,
        wren,
        q);

input [7:0]  address;
input clock;
input [7:0]  data;
input wren;
output [7:0]  q;
```



Page 39

Using this component



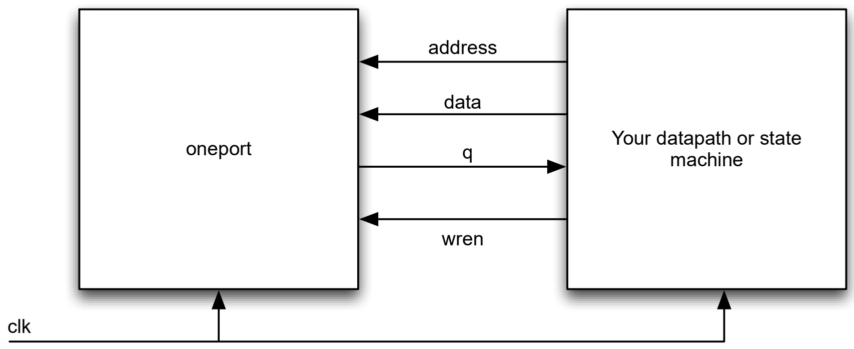
In your datapath or state machine, you can do a write as follows:

```
address <= 8'b011011000;
data <= 8'b100111000;
wren <= 1'b1;
```

Timing is very important

Page 40

Using this component



In your datapath or state machine, you can do a read as follows:

```
address <= 8'b011011000;
wren <= 1'b0;
wait for up to two cycles
value <= q;
```

Timing is very important

Page 41

In general, the more ports, the more simultaneous reads and writes you can do.

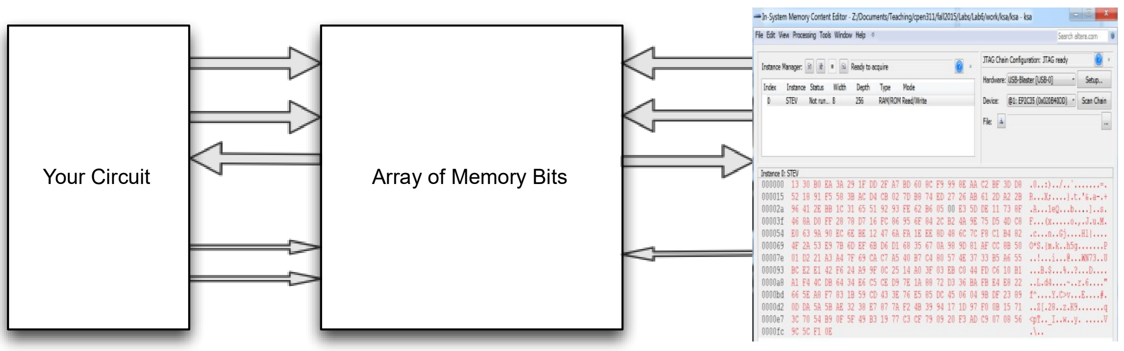
Should you always just use dual-port memories?

Not necessarily:

1. In a custom chip, dual-port memories are a bit slower and larger (not an issue in an FPGA because they are already “there”)
2. You might want to use the other port for something else
3. In some algorithms, it just might not help…

Using other port for something else:

In the lab, you might find it useful to connect one port to the "In-System Memory Content Editor"…


Your Circuit
Array of Memory Bits
In-System Memory Content Editor - Z:/Documents/Teaching/cpen311/fall2015/Labs/Lab6/work/ksa/ksa - ksa
File Edit View Processing Tools Window Help
Search altera.com
Instance Manager:
Ready to acquire
JTAG Chain Configuration: JTAG ready
Hardware: USB-Blaster [USB-0]
Setup...
Device: @1: EP2C35 (0x020B40DD)
Scan Chain
File:

Index	Instance	Status	Width	Depth	Type	Mode
0	STEV	Not run...	8	256	RAM/ROM	Read/Write

Instance 0: STEV
```
000000  13 30 B0 EA 3A 29 1F DD 2F A7 BD 60 8C F9 99 8E AA C2 BF 3D D8  .0..:)../..`......=.
000015  52 18 91 F5 58 3B AC D4 CB 02 7D B8 74 ED 27 26 AB 61 2D A2 2B  R...X;....}.t.'&.a-.+
00002a  96 41 2E BB 1C 31 65 51 92 93 FE 62 B6 05 00 E3 5D DE 11 73 8F  .A...1eQ...b....]..s.
00003f  46 8A D0 FF 28 78 D7 16 FC 86 95 6F 84 2C B2 4A 9E 75 D5 4D C8  F...(x.....o.,.J.u.M.
000054  E0 63 9A 90 EC 6E BE 12 47 6A FA 1E EE 8D 48 6C 7C F8 C1 B4 82  .c...n..Gj....Hl|....
000069  4F 2A 53 E9 7B 6D EF 6B D6 D1 68 35 67 0A 98 9D 81 AF CC 8B 50  O*S.{m.k..h5g.......P
00007e  01 D2 21 A3 A4 7F 69 CA C7 A5 40 B7 C4 80 57 4E 37 33 B5 A6 55  ..!...i...@...WN73..U
000093  BC E2 E1 42 F6 24 A9 9F 0C 25 14 A0 3F 03 EB C0 44 FD C6 10 B1  ...B.$...%..?...D....
0000a8  A1 F4 4C DB 64 34 E6 C5 CE D9 7E 1A 88 72 D3 36 BA FB E4 E8 22  ..L.d4....~..r.6....."
0000bd  66 5E A8 F7 83 1B 59 CD 43 3E 76 E5 85 DC 45 06 04 9B DF 23 89  f^....Y.C>v...E....#.
0000d2  0D DA 5A 5B AE 32 38 E7 87 7A F2 4B 39 94 17 1D 97 F0 0B 15 71  ..Z[.28..z.K9.......q
0000e7  3C 70 54 B9 0F 5F 49 B3 19 77 C3 CF 79 09 20 F3 AD C9 07 08 56  <pT.._I..w..y. .....V
0000fc  9C 5C F1 0E                                                     .\..
```

Useful for debugging because you can see what is in your memory
(also when we test your implementation!)

More details on this tool in the lab text

You can have as many memories in your system as you like
Accesses to memories can be in parallel

Often, each major data structure is mapped to a different memory

Important take away: limited read and write ports adds constraints for scheduling. Must take these into account when designing a circuit that talks to memory.

Page 45

45

Inferred Memories

In Verilog, you can use the array construct to describe a memory

```
  reg [7:0] mem [127:0];
```

You can then describe a read operation behaviourally:

```
  some_variable <= mem[i];
```

You can describe a write operation behaviourally:

```
mem[i] <= some value
```

where i is a variable or signal and contains the address you want to read

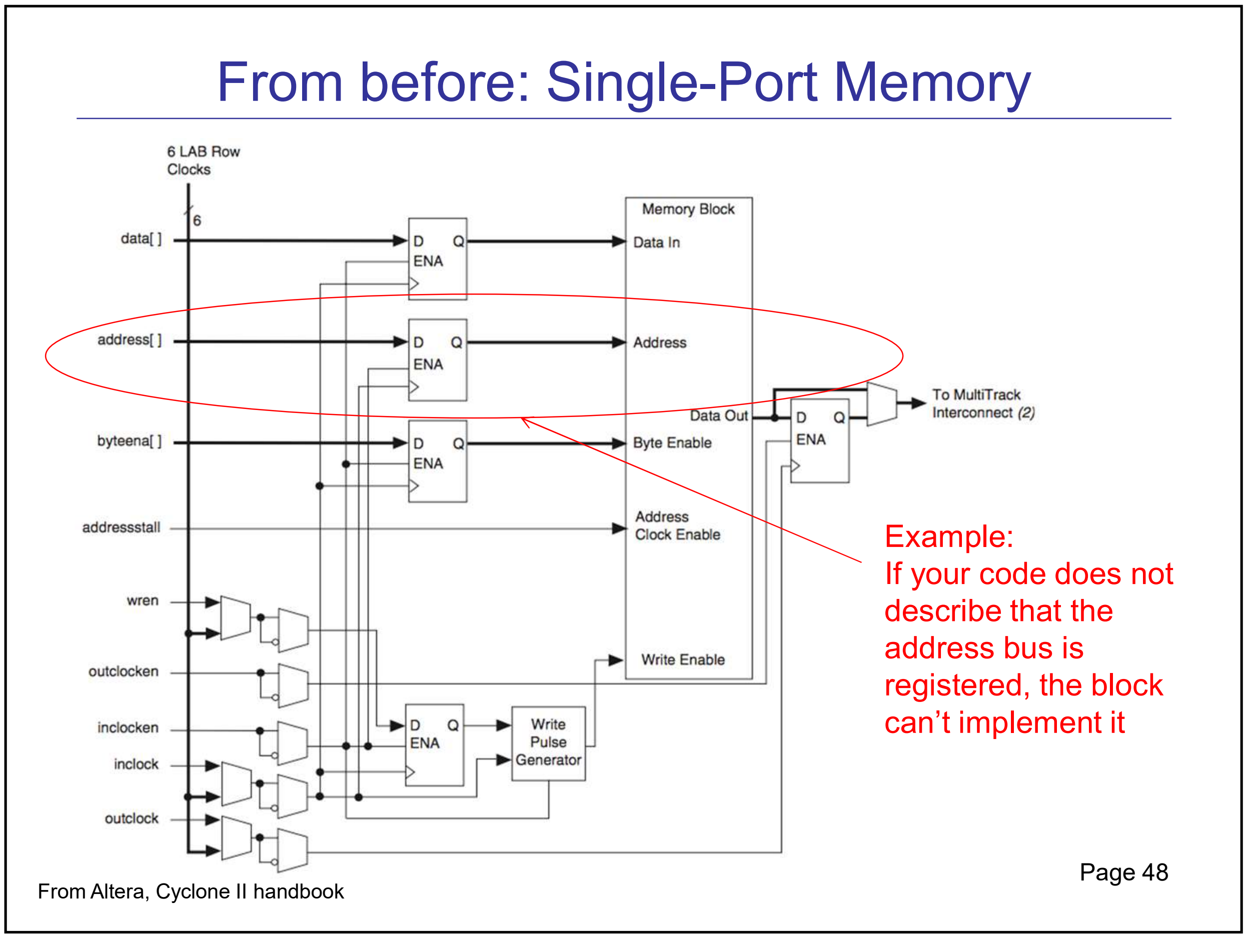
Inferred Memories

But how are these read/write operations included in your code?

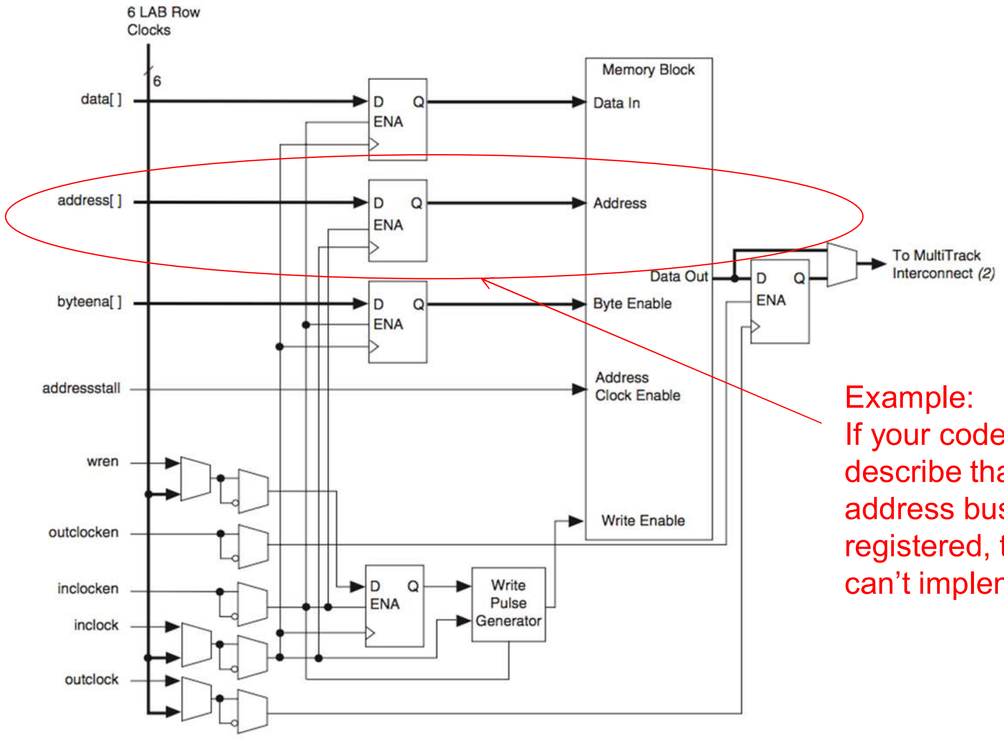
- Are they in a sequential block?
- Does there have to be any relation between read and write?
- How many read and writes are allowed?

The challenge is that the manner in which you include these operations in your code dictates the timing that Quartus Prime has to provide. The embedded RAM blocks **may not be able** to implement the timing that you describe.

- Simple example: if you specify three reads in one cycle, the embedded RAMs won't be able to do it.
- Another example: if your address is not registered, the embedded RAMS won't be able to do it.
- If it does not, Quartus will map it to flip-flop memories



From before: Single-Port Memory



Example:
If your code does not describe that the address bus is registered, the block can't implement it

From Altera, Cyclone II handbook

Page 48

Inferred Memories

Coding rules are well documented in Quartus Prime Handbook
(which you can find on the internet)

Intel Quartus Prime Pro Edition User Guide: Design Recommendations
1.4.1 Inferring RAM functions from HDL Code
https://www.intel.ca/content/www/ca/fr/programmable/documentation/sbc1513987577203.html#mwh1409959579917

WARNING: other tool / FPGA device vendors have different guidelines

- Contains a number of templates that will synthesize to embedded block RAMs

We will not go through the details (read Handbook)

In general, Quartus may find it difficult to identify embedded memories in your logic. Sometimes it will use **flip-flop memories.**

Suggestion #1:
Design your memory as a separate component, following the Handbook's "inferred memory" rules.

Suggestion #2:
If using multiple FPGA vendors (eg, Xilinx and Intel), they both have VHDL/Verilog style guides for inferring memory. **READ THEM** and follow their advice. **When in doubt, test it out!**

Suggestion #3:
Use the Quartus-generated memory blocks as a last resort.

END HERE FOR NOW

Next topic: Scheduling

This is important because the read and write port configuration impacts how you can schedule your algorithm in hardware. In the lab, you will implement an algorithm and scheduling the operations will be a key challenge.

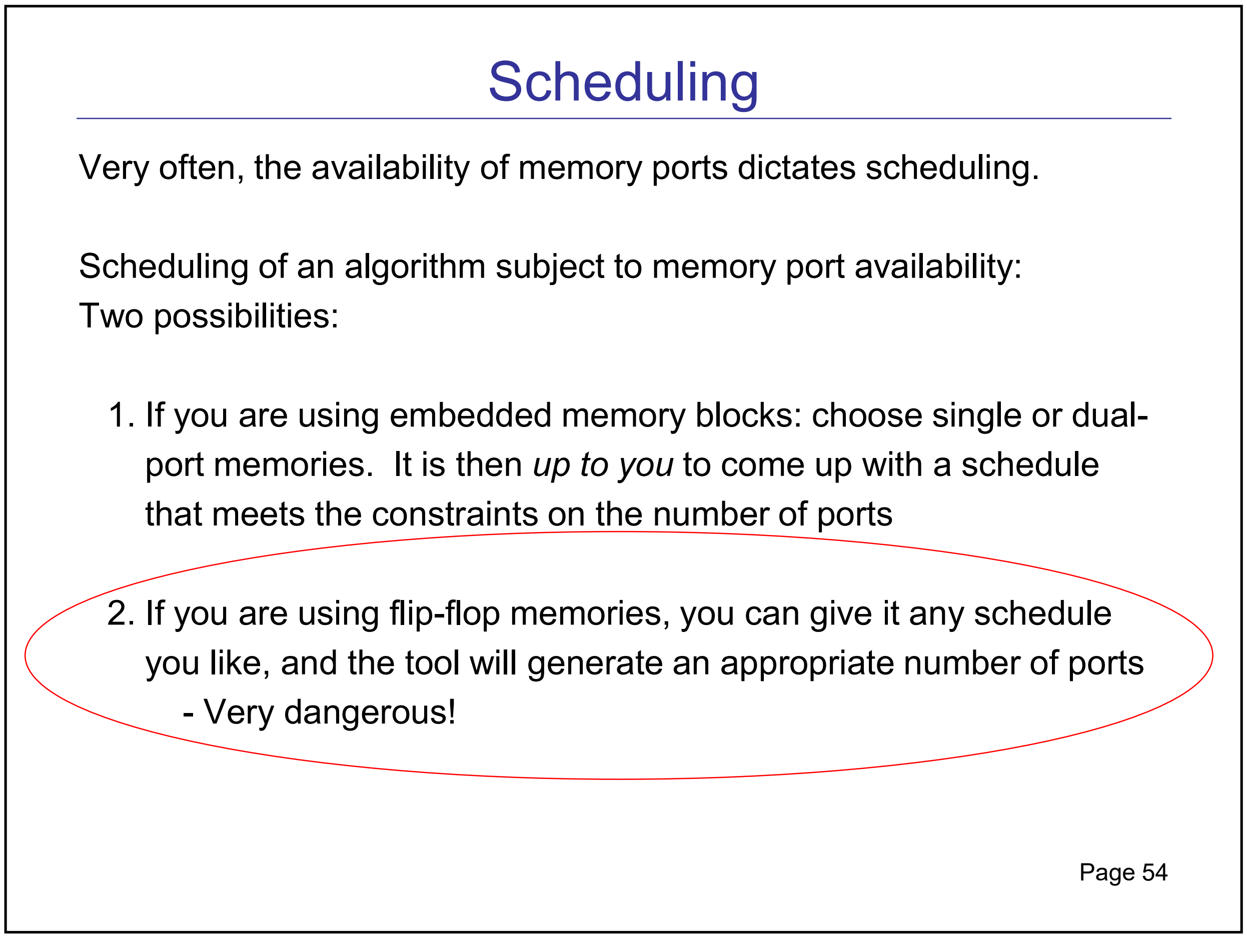
Scheduling

Very often, the availability of memory ports dictates scheduling.

Scheduling of an algorithm subject to memory port availability:
Two possibilities:

1. If you are using embedded memory blocks: choose single or dual-port memories. It is then *up to you* to come up with a schedule that meets the constraints on the number of ports

2. If you are using flip-flop memories, you can give it any schedule you like, and the tool will generate an appropriate number of ports
   - Very dangerous!



Scheduling

Very often, the availability of memory ports dictates scheduling.

Scheduling of an algorithm subject to memory port availability:
Two possibilities:

1. If you are using embedded memory blocks: choose single or dual-port memories. It is then *up to you* to come up with a schedule that meets the constraints on the number of ports

2. If you are using flip-flop memories, you can give it any schedule you like, and the tool will generate an appropriate number of ports
   - Very dangerous!

For the next few slides: given an algorithm, schedule it assuming we are using embedded memory with a single read or write port.

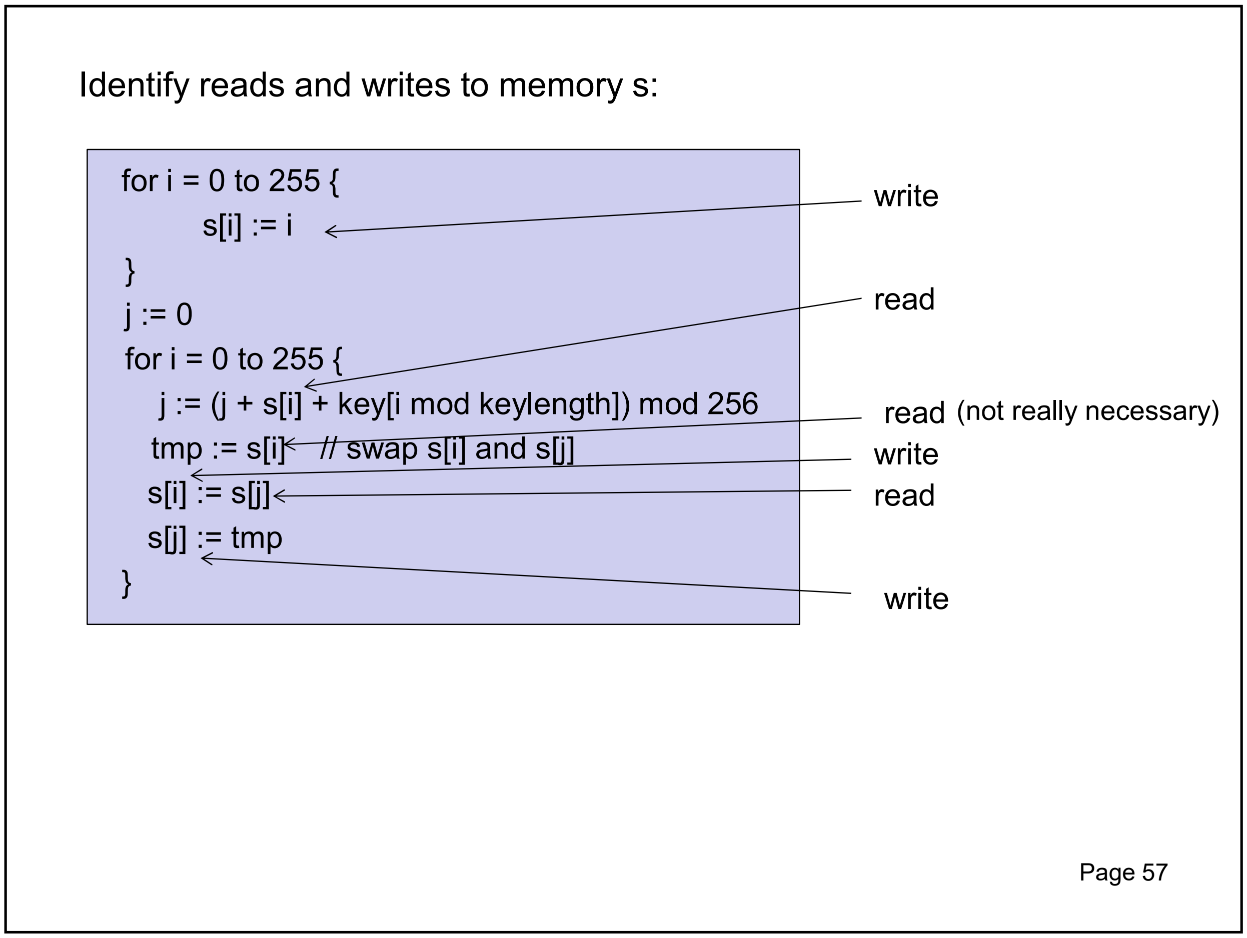
- Can do at most one read or one write each cycle

Scheduling

We will consider this example: (first part of ARC4 encryption):
s is a 256x8 bit memory (256 bytes, each consisting of 8 bits):

```
for i = 0 to 255 {
        s[i] := i
}
j := 0
for i = 0 to 255 {
    j := (j + s[i] + key[i mod keylength]) mod 256
   tmp := s[i]    // swap s[i] and s[j]
   s[i] := s[j]
   s[j] := tmp
}
```

Note: this is **pseudo-code**, not Verilog!



Identify reads and writes to memory s:

```
for i = 0 to 255 {
        s[i] := i
}
j := 0
for i = 0 to 255 {
    j := (j + s[i] + key[i mod keylength]) mod 256
   tmp := s[i]   // swap s[i] and s[j]
   s[i] := s[j]
   s[j] := tmp
}
```

write

read

read (not really necessary)

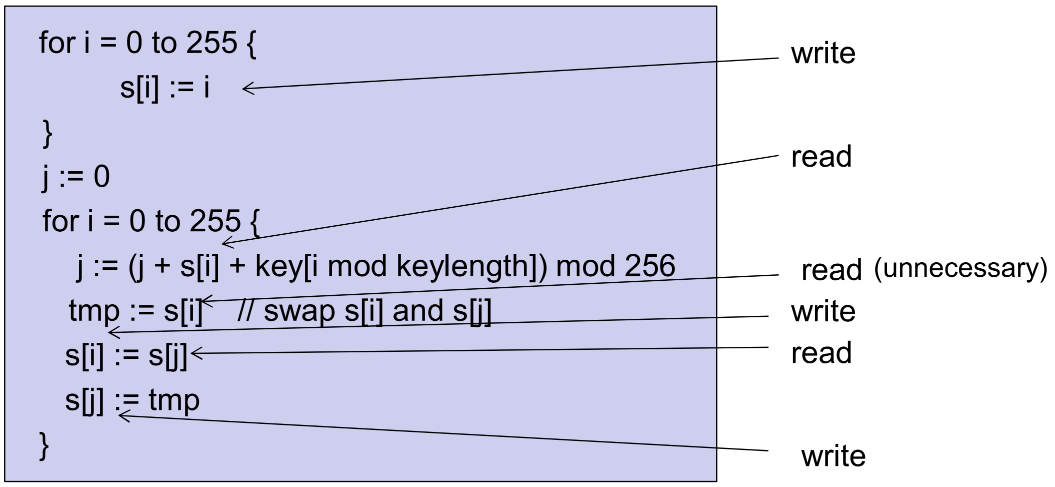
write

read

write

Page 57

57



```
for i = 0 to 255 {
        s[i] := i
}
j := 0
for i = 0 to 255 {
    j := (j + s[i] + key[i mod keylength]) mod 256
   tmp := s[i]    // swap s[i] and s[j]
   s[i] := s[j]
   s[j] := tmp
}
```

Start with the first for loop:

Cycle 0: write 0 to location 0
Cycle 1: write 1 to location 1
Cycle 2: write 2 to location 2
…
Cycle 255: write 255 to location 255

Important point: we can only write to one location per cycle, so we need 256 cycles to complete this loop.

```
for i = 0 to 255 {                                      write
        s[i] := i
}
j := 0                                                  read
for i = 0 to 255 {
   j := (j + s[i] + key[i mod keylength]) mod 256       read (unnecessary)
  tmp := s[i]   // swap s[i] and s[j]                   write
  s[i] := s[j]                                          read
  s[j] := tmp
}                                                       write
```

Now the next for loop:

```
    Cycle 256:  read s[i]   (i==0)
    Cycle 257:  compute j , read s[j]
    Cycle 258:  write s[i]
    Cycle 259:  write s[j]
    Cycle 260:  read s[i]    (i==1)
        …
```

This is just one schedule. There are other ways to do it, but the important point is that there is at most one memory access per cycle since there is only one port

But this schedule MAY OR MAY NOT work! Why?

To understand, let’s look at typical
memory block and timing.

It depends on whether you have
syncronous or asynchronous data-out
for reading memory.

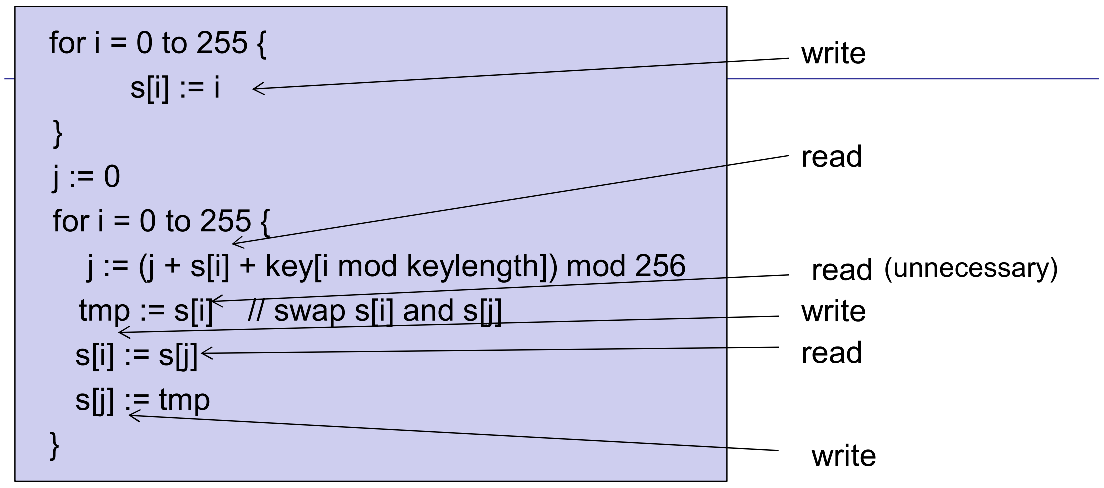
(first, I’ll show you the location of two problems)

```
for i = 0 to 255 {                                        write
        s[i] := i
}                                                         read
j := 0
for i = 0 to 255 {
    j := (j + s[i] + key[i mod keylength]) mod 256        read (unnecessary)
    tmp := s[i]    // swap s[i] and s[j]                  write
    s[i] := s[j]                                          read
    s[j] := tmp
}                                                         write
```

Now the next for loop…

```
    Cycle 256:  read s[i]    (i==0)
    Cycle 257:  compute j , read s[j]
    Cycle 258:  write s[i]
    Cycle 259:  write s[j]
    Cycle 260:  read s[i]    (i==1)
        …
```

In Cycle 256, we read s[i]
In Cycle 257, we hope to use the value. We can't do this if we use sync data-out!



```
for i = 0 to 255 {
        s[i] := i
}
j := 0
for i = 0 to 255 {
   j := (j + s[i] + key[i mod keylength]) mod 256
  tmp := s[i]   // swap s[i] and s[j]
  s[i] := s[j]
  s[j] := tmp
}
```

Now the next for loop…

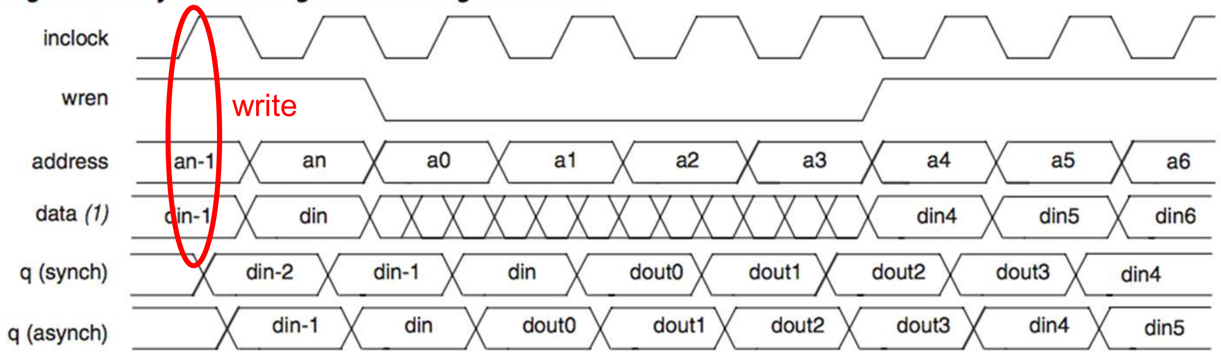
```
    Cycle 256:  read s[i]   (i==0)
    Cycle 257:  compute j , read s[j]
    Cycle 258:  write s[i]
    Cycle 259:  write s[j]
    Cycle 260:  read s[i]    (i==1)
       …
```

In Cycle 257, we read s[j]
In Cycle 258, we hope to use the value to write s[i]. We can't do this if we use sync data-out!

Memory Timing

This is a specific example: the embedded memory blocks on Cyclone II:

Figure 8–7. Cyclone II Single-Port Timing Waveforms



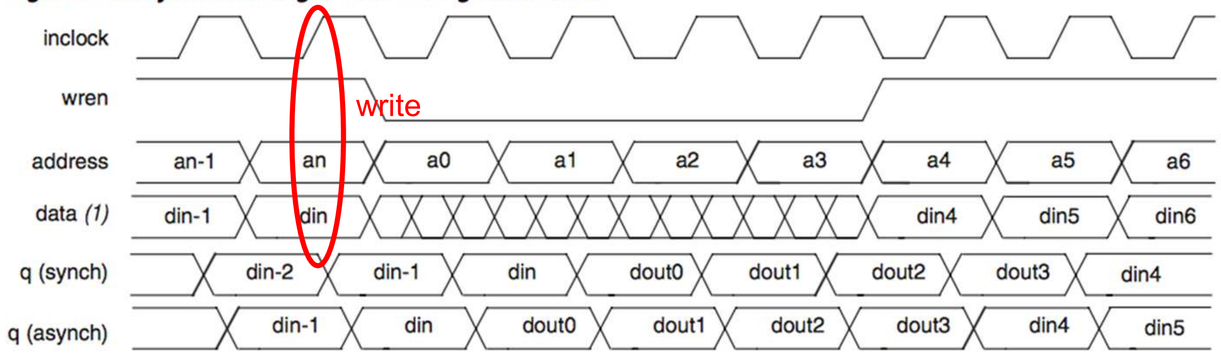
From Altera, Cyclone II handbook

Page 63

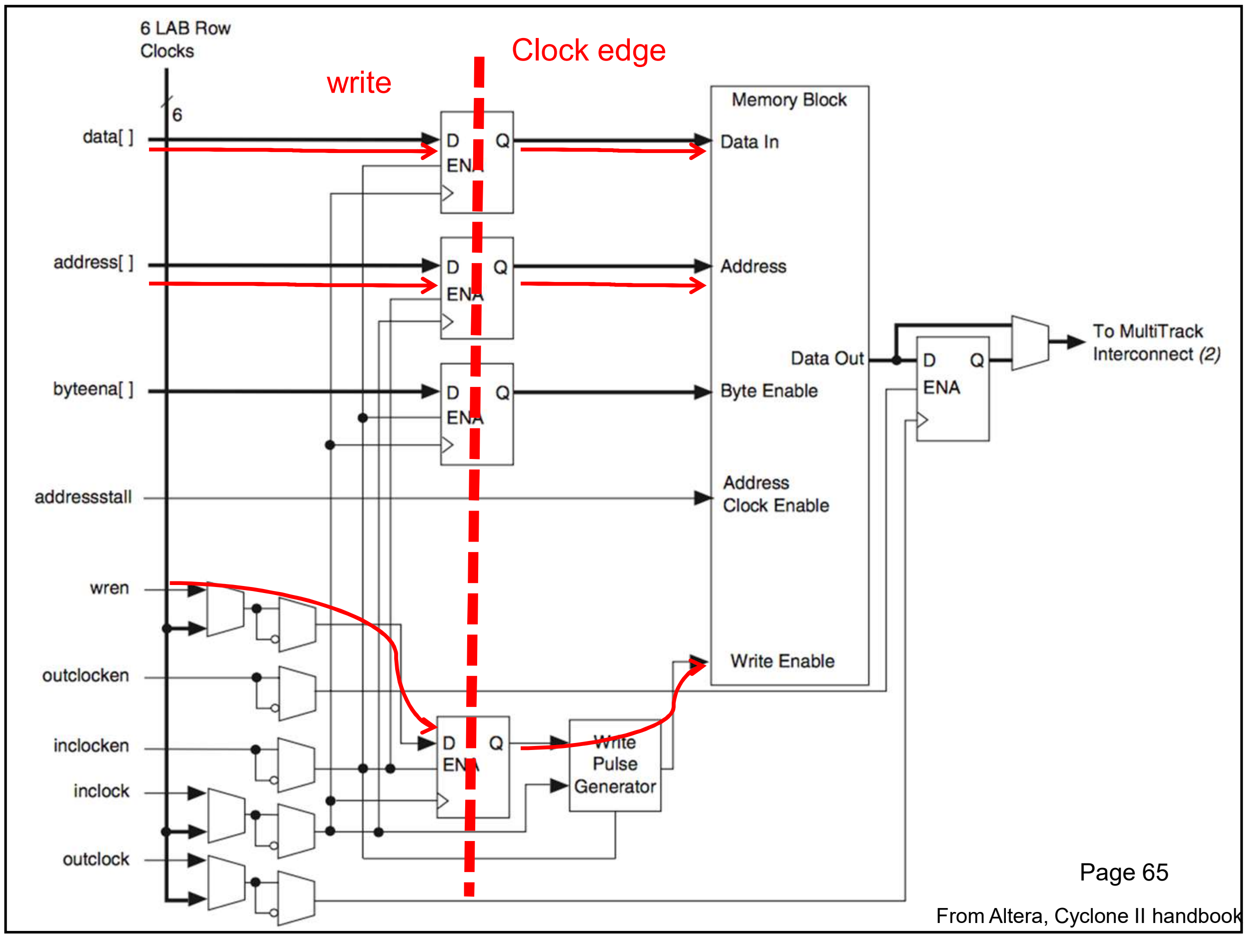
Memory Timing

This is a specific example: the embedded memory blocks on Cyclone II:

Figure 8–7. Cyclone II Single-Port Timing Waveforms

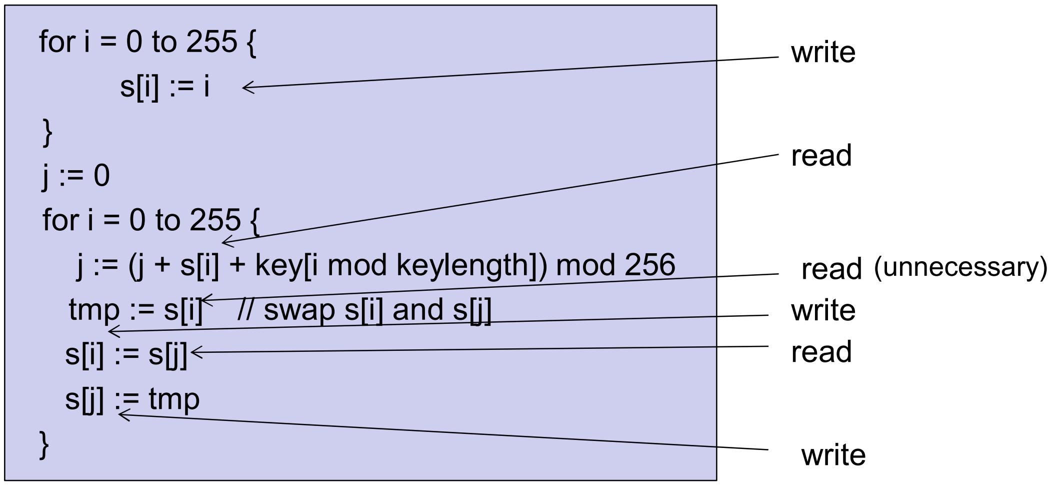


From Altera, Cyclone II handbook



Page 65

From Altera, Cyclone II handbook



```
for i = 0 to 255 {
        s[i] := i
}
j := 0
for i = 0 to 255 {
   j := (j + s[i] + key[i mod keylength]) mod 256
   tmp := s[i]    // swap s[i] and s[j]
   s[i] := s[j]
   s[j] := tmp
}
```

Start with the first for loop:

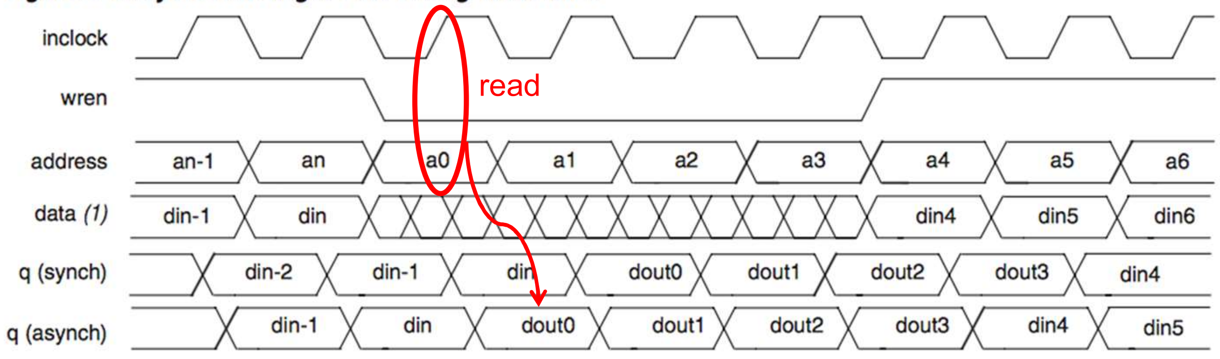
Cycle 0: write 0 to location 0
Cycle 1: write 1 to location 1
Cycle 2: write 2 to location 2
…
Cycle 255: write 255 to location 255

This is all ok. We can issue writes in consecutive cycles

Memory Timing

This is a specific example: the embedded memory blocks on Cyclone II:

Figure 8–7. Cyclone II Single-Port Timing Waveforms

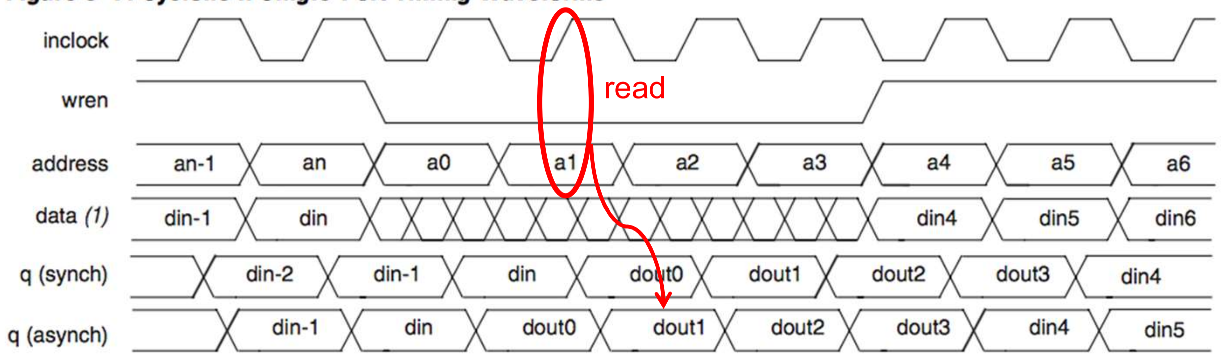


From Altera, Cyclone II handbook

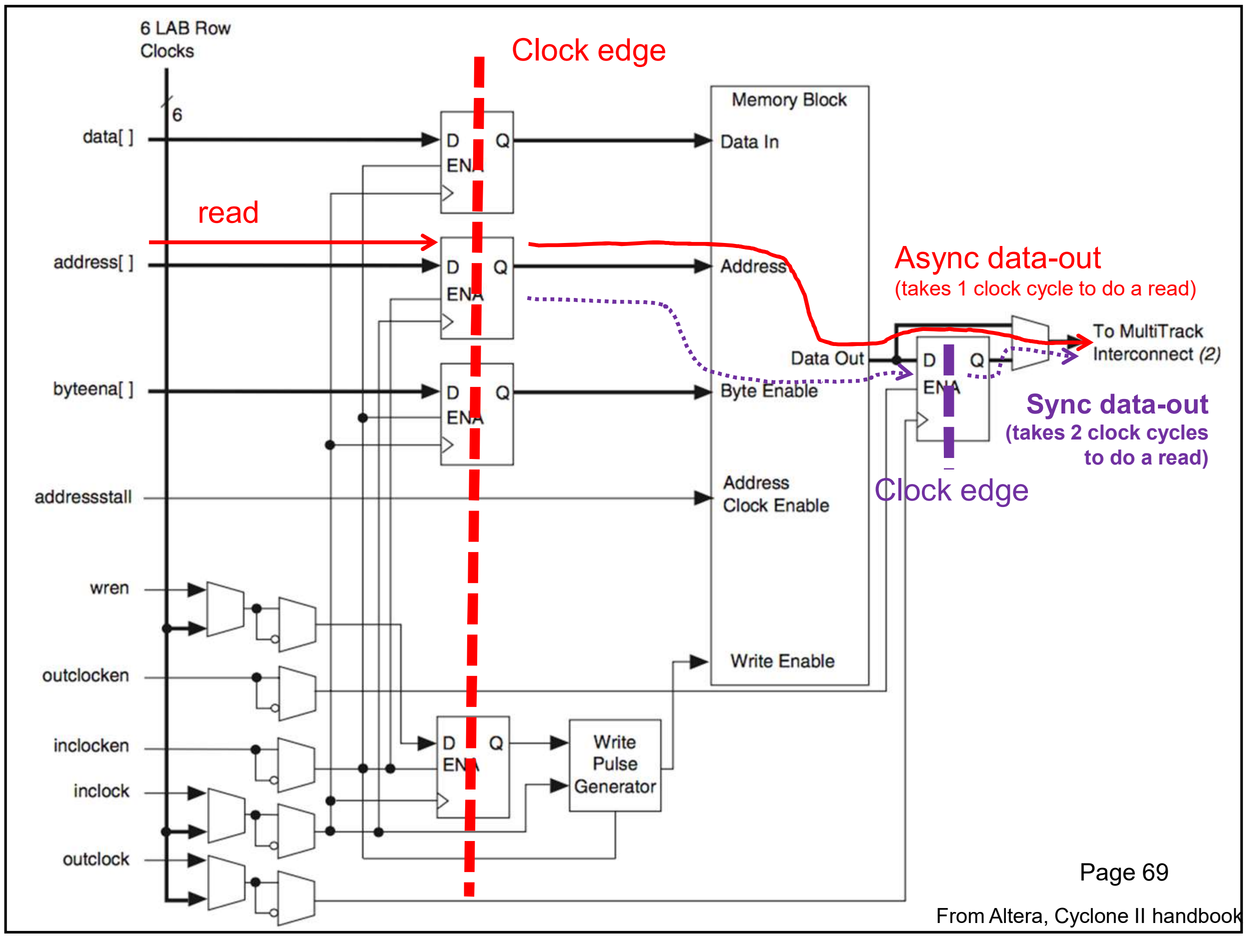
Memory Timing

This is a specific example: the embedded memory blocks on Cyclone II:

Figure 8–7. Cyclone II Single-Port Timing Waveforms



From Altera, Cyclone II handbook

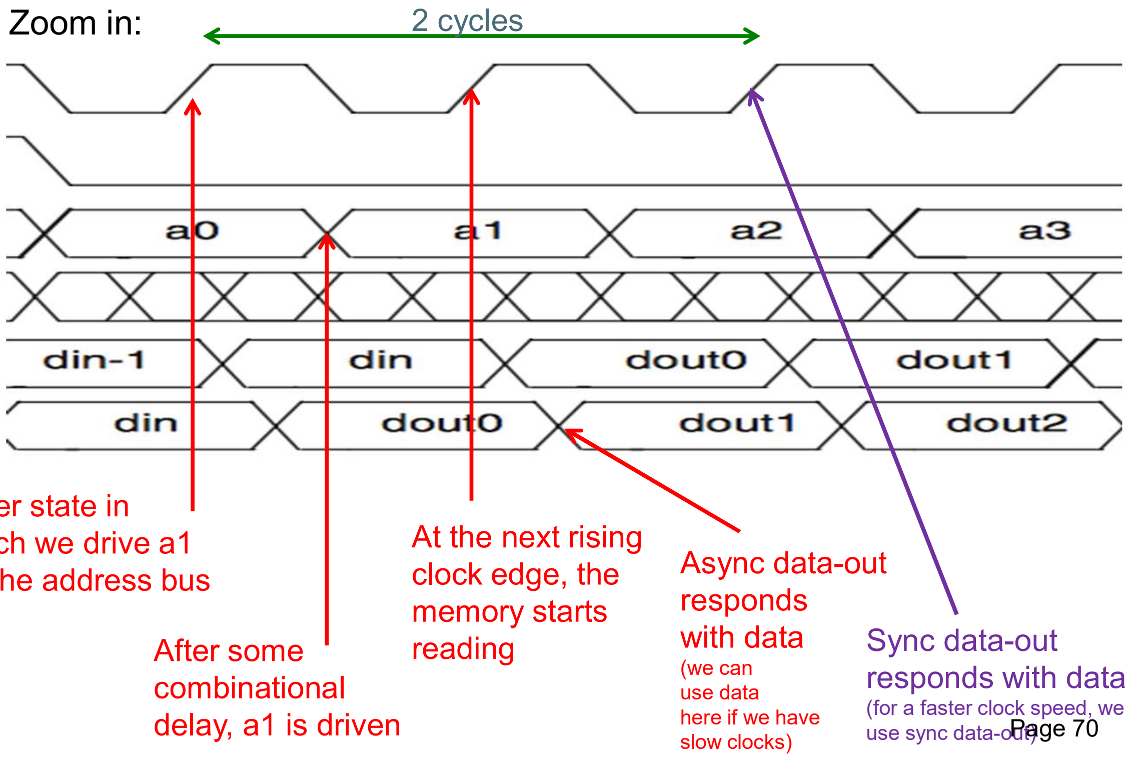


Page 69

From Altera, Cyclone II handbook

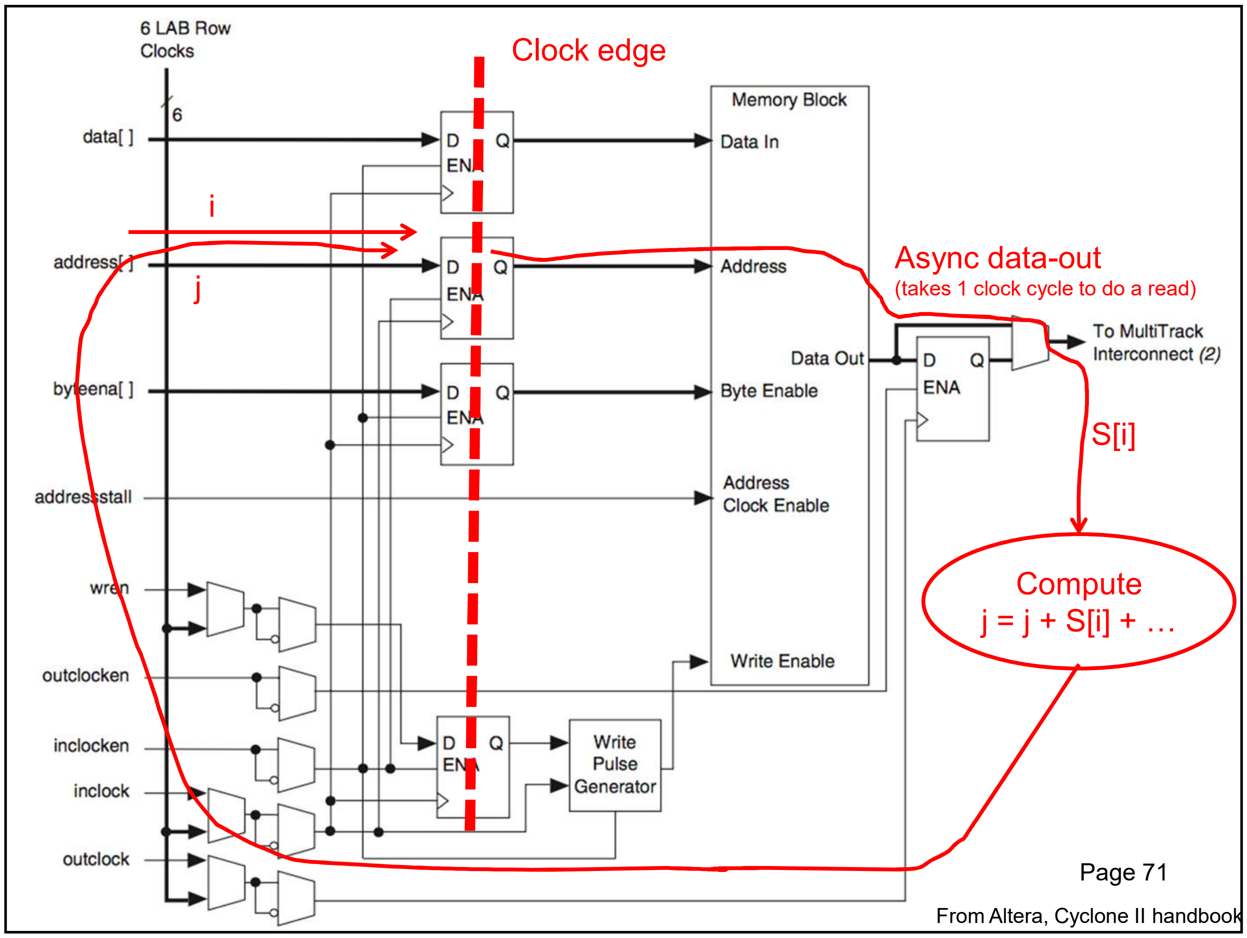
Memory Timing

Zoom in:



2 cycles
a0
a1
a2
a3
din-1
din
dout0
dout1
din
dout0
dout1
dout2
Enter state in which we drive a1 on the address bus
After some combinational delay, a1 is driven
At the next rising clock edge, the memory starts reading
Async data-out responds with data
(we can use data here if we have slow clocks)
Sync data-out responds with data
(for a faster clock speed, we use sync data-out)

Page 70


6 LAB Row Clocks
6
Clock edge
data[ ]
address[ ]
byteena[ ]
addressstall
wren
outclocken
inclocken
inclock
outclock
i
j
D
Q
ENA
Memory Block
Data In
Address
Data Out
Byte Enable
Address Clock Enable
Write Enable
Write Pulse Generator
Async data-out
(takes 1 clock cycle to do a read)
To MultiTrack Interconnect (2)
S[i]
Compute
j = j + S[i] + …
Page 71
From Altera, Cyclone II handbook

71

So what does this mean for our example schedule?

I’ll show 2 problems…

```
for i = 0 to 255 {
        s[i] := i
}
j := 0
for i = 0 to 255 {
    j := (j + s[i] + key[i mod keylength]) mod 256
   tmp := s[i]    // swap s[i] and s[j]
   s[i] := s[j]
   s[j] := tmp
}
```

write

read

read (unnecessary)

write

read

write

Now the next for loop…

```
    Cycle 256:  read s[i]    (i==0)
    Cycle 257:  compute j , read s[j]
    Cycle 258:  write s[i]
    Cycle 259:  write s[j]
    Cycle 260:  read s[i]    (i==1)
        …
```

In Cycle 256, we read s[i]
In Cycle 257, we hope to use the value. We can't do this if we use sync data-out!

```
for i = 0 to 255 {                                       write
        s[i] := i
}
j := 0                                                   read
for i = 0 to 255 {
   j := (j + s[i] + key[i mod keylength]) mod 256        read (unnecessary)
  tmp := s[i]   // swap s[i] and s[j]                    write
  s[i] := s[j]                                           read
  s[j] := tmp
}                                                        write
```

Now the next for loop…

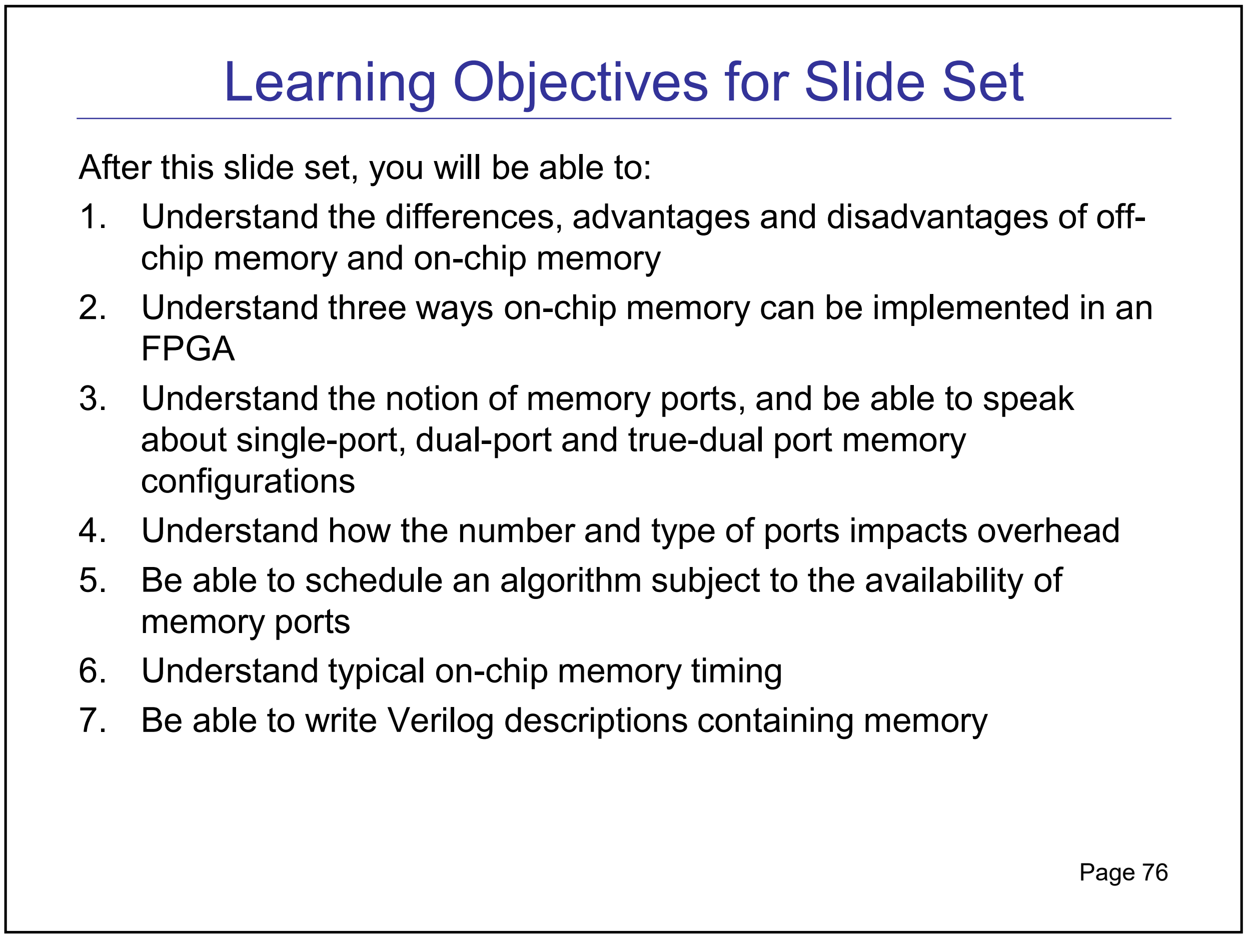
```
    Cycle 256:  read s[i]   (i==0)
    Cycle 257:  compute j , read s[j]
    Cycle 258:  write s[i]
    Cycle 259:  write s[j]
    Cycle 260:  read s[i]    (i==1)
        …
```

In Cycle 257, we read s[j]
In Cycle 258, we hope to use the value to write s[i]. We can't do this if we use sync data-out!

TL;DR

be very careful about modules that use memory

- can’t write to same port multiple times
  - if you do and use implicit memory instances, you will get “memories” from FFs
- can’t load (these) memories within one cycle
- need to design your memory client carefully



Learning Objectives for Slide Set

After this slide set, you will be able to:

1. Understand the differences, advantages and disadvantages of off-chip memory and on-chip memory
2. Understand three ways on-chip memory can be implemented in an FPGA
3. Understand the notion of memory ports, and be able to speak about single-port, dual-port and true-dual port memory configurations
4. Understand how the number and type of ports impacts overhead
5. Be able to schedule an algorithm subject to the availability of memory ports
6. Understand typical on-chip memory timing
7. Be able to write Verilog descriptions containing memory